

NEURON DATA



OPEN INTERFACE

Technical Overview

Neuron Data

Open Interface™

The Tool for Building
Portable Graphical User Interfaces
Across All Windowing Standards

OSF/Motif • Open Look • Microsoft Windows • Macintosh • PM

Copyright © 1991 by Neuron Data Inc.
All rights reserved.

NEURON DATA OPEN INTERFACE™

Technical Overview

Part 1 • Introduction	1
A New GUI Order.....	1
The Origin of Open Interface.....	2
Part 2 • Design Overview	3
Design Philosophy	3
Building Open Interface Applications — An Overview	4
The Architecture of Open Interface.....	5
The Superset Approach	6
Part 3 • Developing with Open Interface	7
Open Editor™	7
Resource Files.....	8
Integrating Application Code — The Notification Model.....	9
Customizing the C Templates	11
Compiling & Linking an Open Interface Application.....	12
Part 4 • Specifying the Look & Feel	13
Setting the Look & Feel Environment Variable.....	13
Porting Applications	14
Part 5 • The Widgets & their API	15
Example 1: The Text Edit.....	15
Example 2: The Table/List Box.....	15
Example 3: The Browser Widget.....	19
Menus and other Widgets in the Open Interface Toolkit.....	21
Part 6 • Integrating Open Interface & Databases	23
A Simple Example Using Open Interface and Sybase™.....	23
Porting the Database Front-end Application.....	27
Database Access with Open Interface and NEXPERT Object.....	27
Part 7 • Extensibility & Custom Widgets	29
The Inheritance Mechanisms of Open Interface.....	30
The Drawing Primitive API.....	31
The Initialization Routines.....	33
Part 8 • Questions & Answers	35
General.....	35
Localization & Multi-byte Character Support.....	37
Microsoft Windows	38
Presentation Manager.....	38
Macintosh.....	38
Application.....	39

Part 1 • Introduction

Neuron Data Open Interface™ is designed to radically change the economics of developing applications with graphical user interfaces (GUIs) for any standard windowing system. Open Interface comprises a set of ANSI C libraries and an interface layout tool with which developers can build graphical user interfaces which are portable across all major windowing, operating system and hardware standards. These include OSF/Motif™, OPEN LOOK™, Presentation Manager™, Microsoft Windows 3.0™ and the Macintosh™.

This paper offers a technical overview of the product, describing its software architecture, its features and the process of developing graphical applications with Open Interface.

This paper has eight parts. Part 1 offers a general introduction to the new reality of GUI programming with Open Interface. Part 2 explains the software architecture and design philosophy of Open Interface. Part 3 describes each step of the process of developing an Open Interface application. Part 4 explains how to port an Open Interface application and specify its look and feel such that it conforms to the look and feel of any native windowing system. Part 5 surveys the Open Interface toolkit and API (Application Programming Interface). This section presents code examples for simple applications which use the text edit, table and browser widgets. Part 6 explains how to use Open Interface as a database front-end tool, and contains example code showing how to integrate Open Interface with the Sybase DB-Library™. Part 7 describes the process of adding new custom widgets to the Open Interface toolkit. Part 8 is an Open Interface question and answer section.

A New GUI Order

Open Interface offers a totally new opportunity for software developers, corporate developers, hardware vendors and system integrators: the ability to develop consistent user interfaces for any platform, under any look and feel.

Widgets should be the same across all platforms. When it comes to interfaces, each platform is, in the end, a machine to somehow light up pixels. It is this kind of full transparency that Open Interface provides. Where one once needed to hire and train programmers for each windowing and operating system, and then wait to see what was produced, these times are now behind. The economics are revolutionized.

Intuitively, this is how things should have been in GUI computing since the beginning. However, the idiosyncrasies of each manufacturer, software vendor or consortium has, over time, created several standards instead of a single one. The result is a nightmare, not just for software vendors, but for corporations and large organizations as well.

The evolution of GUI software necessitates a tool which makes an abstraction of the differences among these windowing systems, while providing the best of each of these existing standards across all platforms. This is what Open Interface does.

Open Interface is a multi-standard and multi-platform GUI development environment. Open Interface offers complete GUI portability across the following windowing systems:

OSF/Motif OPEN LOOK Macintosh Microsoft Windows Presentation Manager
--

Open Interface makes an abstraction of the often arbitrary differences among these windowing systems to provide a unified development continuum for GUIs across the spectrum of standard platforms.

The new order entails being able to use any platform for the development of a professional graphical user interface and to port it to any other platform by simply recompiling and linking on the target platform, without a perfor-

mance penalty. Such a technology step was necessary to allow software vendors and corporate developers to build applications in step with and in relation to their business environment.

The Origin of Open Interface

In 1985, Neuron Data launched NEXPERT OBJECT™, the leading software tool for developing and delivering industrial expert system applications. NEXPERT, initially developed on the Macintosh, is well-known for its highly-graphical, intuitive development environment. NEXPERT's inference engine was written in C and could be easily ported to other hardware platforms and operating systems. However, when, in 1986, it came to moving NEXPERT's graphical development environment to other windowing systems, Neuron Data's software engineers were faced with the same challenge which so many developers face today.

Rather than re-coding the interface for each of the various windowing systems which were then emerging (Windows 1.0 and X10), after examining the strengths and weaknesses of all

of the different graphical environments and toolkits, work was begun on an in-house tool, Open Interface, to make this process more efficient.

From its inception, Open Interface was a tool built by developers for developers. Open Interface was not created in a vacuum — it was designed to support the delivery of a very successful commercial software product across a wide range of windowing and operating systems. To ensure the success of NEXPERT OBJECT, Neuron Data's engineers needed a high-performance, portable GUI development tool that was easy to program, had a powerful widget set, conformed to the native look and feels, and which had only a minimal memory impact.

Open Interface has been continually refined and enhanced over the past five years, and has enabled Neuron Data to port the NEXPERT graphical development environment to more than 30 platforms. In 1991, Open Interface was made commercially available as a developer tool for building portable graphical interfaces.

Part 2 • Design Overview

Design Philosophy

Open Interface was designed to support five key goals:

- **True portability** - Open Interface enables developers to build graphical interfaces which are always 100% portable. This does not come at the expense of interface functionality. The restrictions imposed by the "least common denominator" approach were deemed unacceptable; this only provided portability for the very limited set of widgets which fall in the intersection of the various native toolkits. All of the widgets in the Open Interface toolkit are available on all supported platforms. At the same time, applications developed with Open Interface co-exist with applications written using the native windowing systems and toolkits. Nor are any special or modified window managers required. By implementing a widget set and an event manager on top of the lowest level of the native windowing system, Open Interface ensures that all interface components are both 100% portable and 100% compatible.

- **Full Extensibility** - Although the set of widgets (buttons, list boxes, text edits, etc.) provided by the Open Interface toolkit is extensive, new, custom widgets can be added to extend the toolkit. Furthermore, the Open Interface model of extensibility provides complete portability of these new widgets — *they do not need to be re-implemented for each windowing system*. In this way, developers can extend the toolkit to suit their particular line of business — to add, for example, dial and gauge widgets for real-time process control applications, or statistical graphing widgets for financial applications.

- **Performance** - From its inception, Open Interface has been designed as a tool with which to develop robust, production applications like NEXPERT OBJECT. It was essential that portability and extensibility not come at the expense of performance — both in terms of drawing speed and memory overhead.

Compared to applications built with the Motif, OPEN LOOK and Xt Intrinsics toolkit libraries, the responsiveness of Open Interface applications is usually superior. This is due largely to the fact that, unlike Motif and OPEN LOOK, an X Windows window is not opened for each Open Interface widget, and also that Open Interface caches and shares its low-level resources (like fonts, colors, etc.) in a very efficient manner. This performance improvement is particularly noticeable on X-server terminals. On the Macintosh, Windows and PM windowing systems, the responsiveness of applications built with Open Interface is always comparable to a similarly-behaving, native application.

Code overhead is minimal. The executable size of a small Open Interface application is anywhere from 300K on a Macintosh, to 700K on a Unix machine. Given the object-oriented implementation of Open Interface, for any given application, a majority of the interface code is reused. Thus, as an application's interface complexity grows by an order of magnitude, the size of its executable may only increase by fifty percent. In effect, the size of the Open Interface "engine" (library) always remains the same — it is only the extra code to implement the application which increases the size of the executable. In addition, wherever possible, the Open Interface libraries are implemented in a shared format, either as DLLs (Dynamically Linked Libraries) on the PC, as shared libraries under Unix, or as shareable images under VMS.

- **Powerful API** - Professional graphical user interface programming is a tough task — witness the huge volume of documentation for Motif or Windows. By choosing the right API to the Open Interface libraries, the tradeoffs between the simplicity, power, expressiveness and fine control provided by the toolkit have been optimized. With Open Interface, rich interface functionality can be implemented with a minimal number of function calls. Early customer feedback has further corroborated the value of the careful delineation of the Open

Interface API.

• **Powerful tools** - To boost developer productivity, Open Interface provides not just the right API, but also the right tool to generate interfaces and to integrate them with the underlying application functionality. Open Editor™, is a fully-featured, WYSIWYG interface editor, used to interactively layout graphical user interfaces.

Open Editor provides a complete set of window, widget and resource editors to design, revise and organize interface components. Open Editor generates the Open Interface resource files, ANSI C source code templates, and the makefiles needed to implement the application interface.

Building Open Interface Applications — An Overview

Open Interface consists of Open Editor™, a WYSIWYG interface development environment, and the platform-specific Open Interface runtime libraries.

an application's GUI. Simple point-and-click operations are used to design and edit windows, menus, list boxes, buttons, etc. Open Editor then generates three files: resource files, C templates and makefiles, as shown in Figure 1.

Once these files have been generated, the developer must customize the C templates to add the interface logic and back-end functionality. Finally, to complete the application the makefile is used to compile the code and to link it to the non-GUI functionality and to the Open Interface libraries.

To port an application to another platform, the resource files and C source code files are transferred to the target environment using a network or some magnetic media, and then recompiled and relinked with the Open Interface libraries for that platform.

"Part 3 • Developing with Open Interface" describes each step of this process in greater detail.

4

Open Editor is used to layout the components of

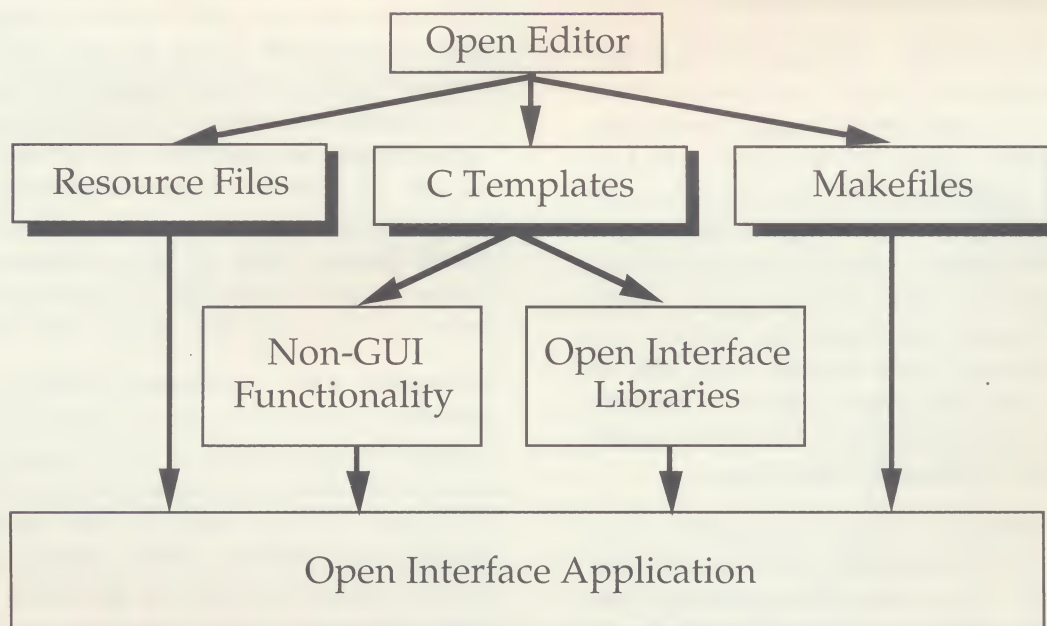


Figure 1: The Open Interface Development Process.

The Architecture of Open Interface

Open Interface, architected on the principle of a layered Application Programming Interface (API), shields the developer from the variances of the underlying windowing and operating systems. All of the procedures and functions in the Open Interface libraries are available on all of the platforms and, regardless of the windowing system, behave identically. Thus, 100% portability is always ensured; a single source code file can be compiled and linked on any of the target platforms without any conditional compilation (`#ifdef's`). At the same time Open Interface applications co-exist with native applications and window managers, without any modification to the underlying windowing system.

The Open Interface Virtual Graphics Machine™ library (VGM) is the portable, graphical foundation of Open Interface. It provides a complete set of graphic primitive routines for drawing lines, text and colors which behave identically on every platform. It also maps the events received from the native windowing systems, like mouseclicks or keyboard actions, into a common representation across all platforms.

The Toolkit library contains all of the routines necessary to create, display and manipulate the Open Interface widget set. The behavior of each widget is defined by calls to the Virtual Graphics Machine, relative to events generated by the VGM. As an example, a text button widget can be implemented basically by using two VGM functions: `DRAW_Rect()`, to draw a filled rectangle (the button) and `DRAW_Text()`, to draw a text string (the button label). As both of these VGM functions are completely portable, the text button widget, like all of the widgets in the Open Interface toolkit, is available on every platform. The widgets currently supported in the Open Interface toolkit are:

- Static Text Area
- Multi/Single Line Text Edit
- Scroll Bar
- Scrollable Area
- Scrollable Area Overview
- Panel
- Menu Bar
- Popup (Cascading)
- Push Button
- Radio Button
- Check Box
- Browser
- Browser Overview
- Bitmap
- Icon Button
- Table/List Box

Because the various look and feels are also implemented on top of the portable Virtual Graphics Machine, all of the look and feels are available on every platform. For example, it is

5

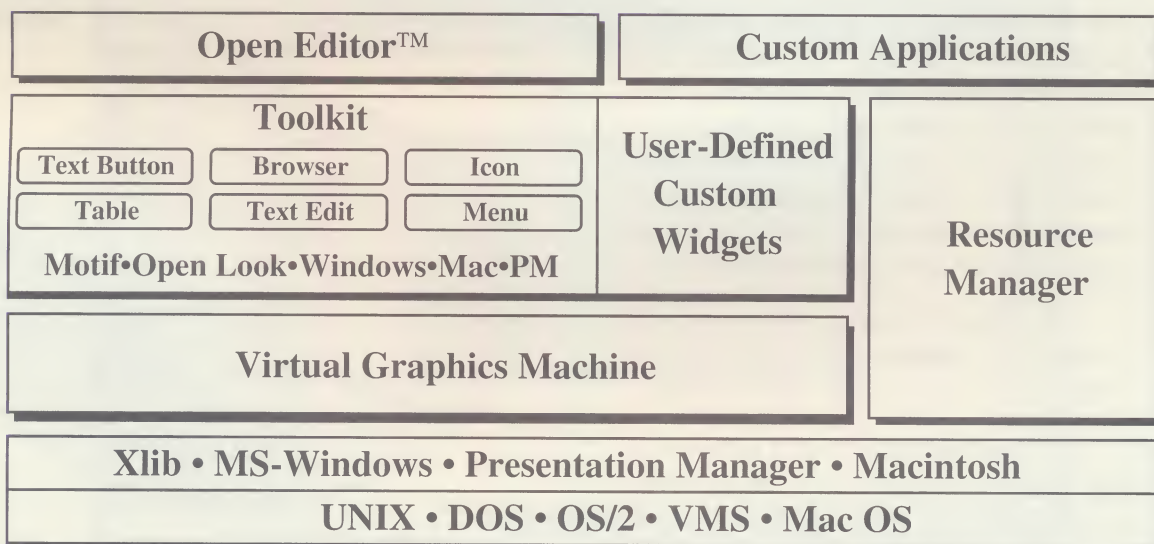


Figure 2: The architecture of Open Interface.

possible to run an Open Interface application on a Macintosh with a Motif look and feel, or to run the same application on a Sun SPARC with an OPEN LOOK look and feel. "Part 4 • Specifying the Look and Feel" covers this topic in more detail.

Although the Open Interface toolkit is a superset of the widgets available in the native toolkits of all of the windowing systems, it can be extended to include new, custom widgets specific to a particular type of application. For instance, an application for real-time manufacturing control may require some kind of gauge or dial widget. By defining this widget's appearance and behavior in terms of calls to the Virtual Graphics Machine, a new widget class can be created and used in any Open Interface application on any platform. It is also possible to tailor the behavior and appearance of a custom widget in order to adhere to the style guide of a specific look and feel. The technical details of extending the toolkit are explained in "Part 7 • Extensibility and Custom Widgets".

There are four other Open Interface libraries:

- The Resource library manages the Open Interface text and binary resource files which describe the interface.
- The Core library is a portable replacement/extension to the standard C Runtime Library; it implements portable memory allocation, portable file I/O, portable data types and arrays and an exception-based error handling model. The Core library also implements the single and multi-byte string manipulation routines needed for Native Language Support (NLS) and for applications using the Kanji character sets.
- The Generic Windows library provides a number of standard windows and dialogs, such as error windows and Yes/No dialogs.
- The Open Editor library is provided so that the interface layout tool can be rebuilt to incorporate new custom widgets.

Wherever possible, all of the above libraries are implemented in a shared format. Under DOS

and OS/2 they are Dynamically Linked Libraries, under Unix they are shared libraries and under VMS they are shareable images.

The Superset Approach

Building a GUI tool that represents a new step in computing required that the ill-fated, "least common denominator" approach to GUI portability be abandoned. This approach only offers portability for the very limited set of widgets which fall in the intersection of the various native toolkits, as illustrated by the dark gray area in Figure 3a below:

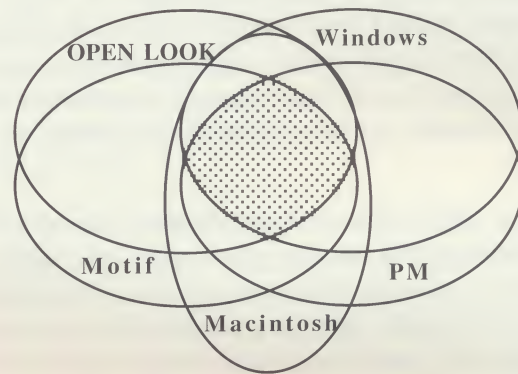


Figure 3a: The Least Common Denominator

Open Interface implements a toolkit on top of a portable Virtual Graphics Machine. This approach offers a superset of all of the widgets available in all of the native toolkits, and can be extended to include new, powerful widgets not offered in any of the toolkits.

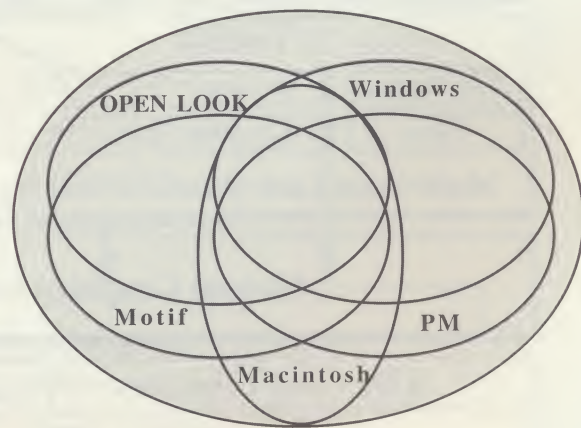


Figure 3b: The Superset of Open Interface

Part 3 • Developing with Open Interface

This section describes each step in the process of developing GUI applications with Open Interface — from creating the interface components with Open Editor to compiling and linking the completed application.

Open Editor™

Open Editor is an interactive, WYSIWYG graphical interface development environment. Using simple point-and-click operations, developers can create and tailor the requisite interface components of any GUI application, from high level resources like windows, menus and widgets, to low-level resources like fonts, cursors and colors. Since Open Editor is built entirely on the Open Interface libraries, it is completely portable and offers identical functional-

ity on every platform. An application can be developed on any of the supported platforms and ported to any other; there is no single development platform of choice.

Within Open Editor, each type of resource, such as windows, widgets, string messages or colors, has its own editor. These are used to customize each instance of a resource. For example:

- Within the window editor, widgets can be selected from the widget palette, and then drawn onto the window with a click-and-drag operation.
- The list box editor is used to specify attributes like background color, row and column sizes, or selection behavior.

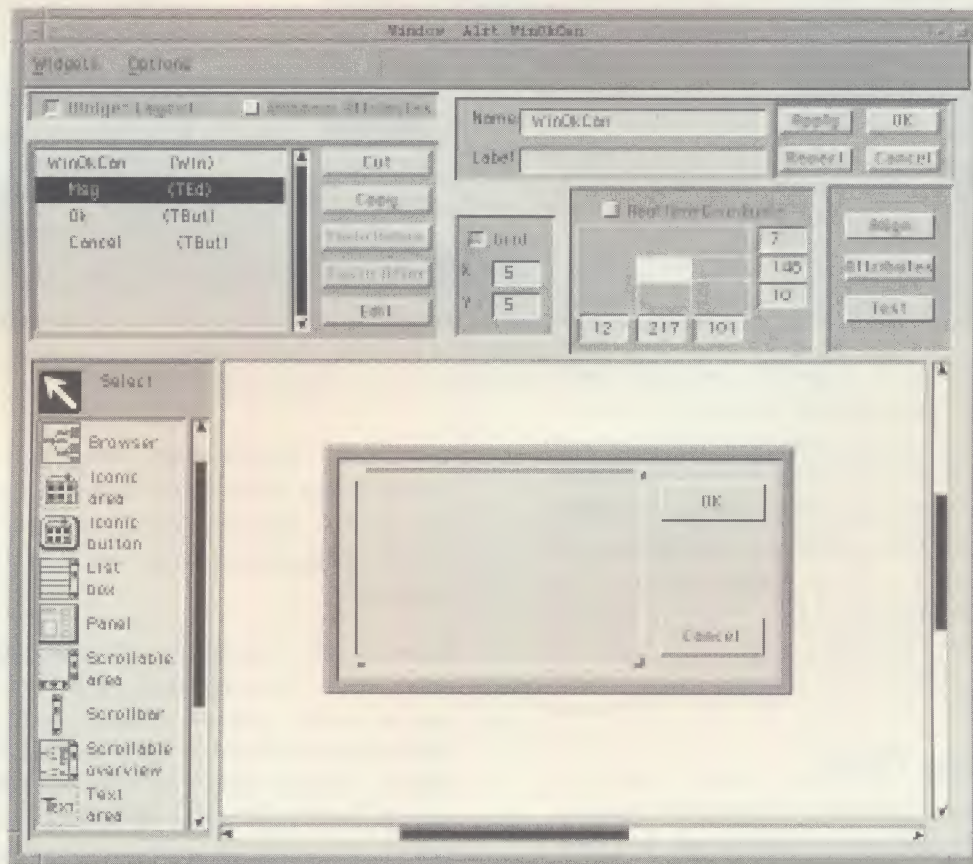


Figure 4: The Window editor of Open Editor™.

- The icon editor has tools to perform complex bitmap manipulations
- The color editor offers a palette from which a particular RGB value can be selected and named.

The Open Editor's Resource Browser is used to organize, edit and display a hierarchy of an application's interface components. This navigation tool gives developers a complete overview of their resource hierarchy, from libraries to low-level resources. Each node in the hierarchy represents an instance of an Open Interface resource, such as a window or a widget. By double-clicking on a node in the Resource Browser, the editor for a particular resource can

ments its behavior. In object-oriented terms, resource files are used to maintain all of the persistent data of each object. For instance, the position and default font of a text edit field would be stored in a resource file, while the data entered into the text edit during runtime would be handled in the application code itself.

Open Interface implements its own portable resource format. To ensure efficiency and portability of resources across platforms, resource files are maintained in both a text (portable) and binary (efficient) format. For a given target platform, an application's text resource files (.rc) are run through Rescomp, the Open Interface Resource Compiler, in order to generate an efficient binary resource file (.dat) used

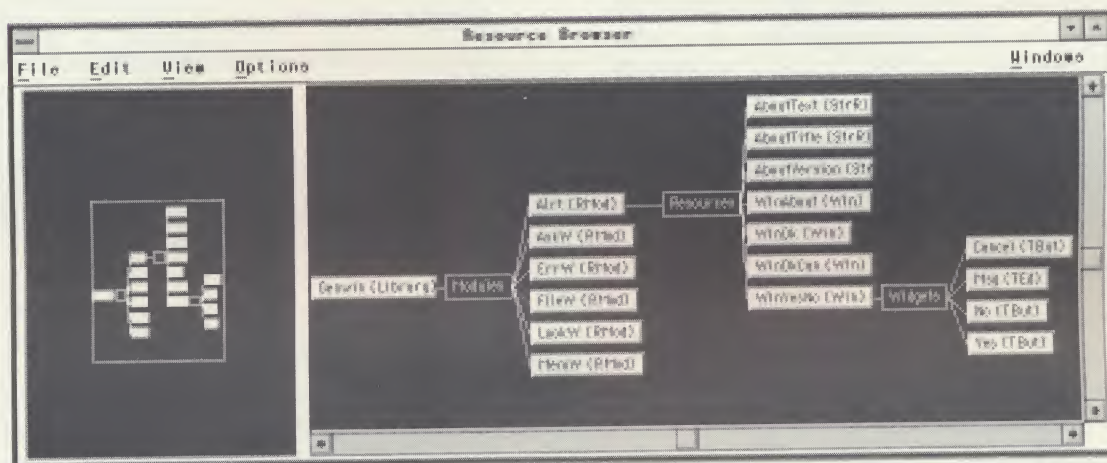


Figure 5: The Open Editor Resource Browser and Overview.

be invoked to edit its attributes.

Once the components of the interface have been defined, Open Editor generates four files: a text resource file (.rc), a binary resource file (.dat), a C template, and a platform-specific makefile. The following sections describe the content of each of these in greater detail.

Resource Files

Open Interface uses the concept of resources in order to modularize the interface code. Resources maintain the display attributes of a graphical object, such as its position, size or color, separately from the C code which imple-

ments its behavior. Because Open Interface resources are not bound to the application code, each user can have a separate binary resource file in which customized user-settings can be maintained.

Since interface attributes are maintained separately from the application source code, they can be easily changed without modifying the underlying executable. Resource files can be modified using Open Editor, or by editing the text resource file. Since all labels, menu items and messages can be stored as string resources, an application can be localized to support any foreign language simply by translating its re-

source files. The associated application source code would not even need to be recompiled. Furthermore, since all of the string manipulation routines in Open Interface support multi-byte characters, applications can be localized to create Kanji (Japanese) applications without any source code modifications.

Open Interface resources are organized into libraries and modules. Usually, one window and all of its associated resources are maintained in one module. These modules are grouped into a resource library and compiled into one binary resource file (.dat).

Code Example 1 shows an extract from an Open Interface text resource file. It shows the resources for a text button and a string message.

```
(TBut.Compile
  Name:      "Mod.Win.QuitButton"
  FgColor:   "TBut.ColorFg"
  BgColor:   "TBut.ColorBg"
  Font:      "Font.Normal"
  Pen:       "Pen.Hole2"
  Pattern:   "Patt.Empty"
  Cursor:    "Curs.Arrow"
  X:         112
  Y:         166
  W:         123
  H:         53
  Label:     "Quit"
  LabelColor: "TBut.ColorText"
  ToggleKeys: "TBut.KeysToggle"
  TextColor: "TBut.ColorText"
)

(StrR.Compile
  Name:      "Mod.Confirm"
  Text:      "Do you want to quit?"
)
```

Code Example 1: A text resource file generated by Open Editor.

Note that some values for a widget may be logically mapped to lower-level resources. In the example above, the background color of the text button is mapped to `TBut.ColorFg`, the default background color for all text buttons. This color resource is maintained in the module `TBut`, where it is assigned a particular RGB color

value. Since Open Interface resources are logically mapped, any changes to the RGB values of `TBut.ColorFg` would be globally propagated to all of the widgets which use this resource. This makes maintaining the interface much easier.

Integrating Application Code — The Notification Model

The C source code templates generated by Open Editor provide the foundation for any Open Interface application. The core of these C templates are the notification procedures (or callbacks), one for each active widget and window in an application. When an event (a mouseclick or keyboard input, for example) occurs within the boundaries of a particular widget, the notification procedure associated with that widget will be called to handle that event. Collectively, notification procedures define an application's presentation logic and link the interface to the application-specific functionality. Code Example 2 shows a template notification procedure generated by Open Editor for a text button.

As part of the window initialization logic, a notification procedure can be registered for each widget in the window and for the window itself. These notifications are dispatched automatically by the Open Interface event loop. Within a notification procedure, calls may be made to the Open Interface libraries to manipulate the interface (to move data from a text edit to a list box cell, for example), or to those libraries which implement some underlying, non-GUI application functionality (to query a database, for example).

Each widget type has its own default notification procedure which implements its default behavior. For example, the default notification procedure for the text edit widget class, `TED_DefNfy()` implements justification, scrolling, resizing, character insertion, selection, etc. This means that for a particular instance of any widget, its notification procedure only needs to trap those events for which the default behavior is not sufficient.


```
static void Mod_QuitButtonNfy(TButPtr tbut, TButNfyEnum code)
{
    ERR_TRACEIN; /* Used by error-handler */
    if (code != TBUT_NFYHIT) { TBUT_DefNfy(tbut, code); ERR_TRACEOUT; }
    /* USER CODE */ /* Add code to handle mouse hit event */
    ERR_TRACEOUT; /* Used by error-handler */
}
```

Code Example 2: The template text button notification routine generated by Open Editor.

For instance, the default behavior of an icon button, is to reverse the colors of the icon while the mouse button is down. In order to implement any other functionality, like having a new window open when a click occurs on the icon button, code must be added to the notification procedure to trap the `IBUT_NFYMOUSECLICK` notification for that particular icon button.

The C templates also contain the `main()` entry point, the window initialization logic and some header information, such as the indices needed

to navigate in a window's widget hierarchy.

The main Open Interface routine shown in Code Example 3 is very straightforward. After the routines to initialize the Open Interface libraries have been called, the application's binary resource file is loaded (in this case `wplib.dat`), the first window is opened and the main event loop is called. This event loop will continue to dispatch notifications to windows and widgets until the application is terminated.

10

```
/* main() entry point */
#include <appub.h>
#include <rllibpub.h>
#include <gwpub.h>

int main (int argc, char** argv)
{
    /* Initialization routines */
    APP_InitFirst();
    GW_LibInit();
    APP_InitLast();

    /* Load the binary resource file */
    RLIB_LoadLibFile("WPLib", "wplib.dat");

    /* Open the first window of the application */
    Mod_WinInit();

    /* Go into the main event loop to dispatch events */
    EVENT_MainLoop();
    return 0;
}
```

Code Example 3: The Open Interface main routine generated by Open Editor.

```

/* Window Initialization Routine*/
void Mod_WinInit()
{
    WinPtr    win;
    PanelPtr  panel;

    /* Load and initialize the window's resources from the module "Mod" */
    win = WIN_LoadInit("Mod", "Win");
    panel = (PanelPtr) win;

    /* Register the notification routine for the Quit button */
    PANEL_SetSubNfyProc(panel, MOD_QUITBUTTON, Mod_QuitButtonNfy);
    WIN_Open(win); /* Display the window */
}

```

Code Example 4: The window initialization routine generated by Open Editor.

In the window initialization procedure shown in Code Example 4, `WIN_LoadInit()` is called to load the resources for the window and all of its widgets from the binary resource file. Then `PANEL_SetSubNfyProc()` is called to register a notification routine for each widget in the window — in this case `Mod_QuitButtonNfy()` is associated to the widget `MOD_QUITBUTTON`. `Mod_QuitButtonNfy()` will be called to handle each event which occurs within the boundaries of the “Quit” text button. Finally, `WIN_Open()` opens and displays the window.

Open Editor generates C code templates by selectively concatenating string resources from the Open Editor resource library. These string resources can be edited to have the templates generated in a different fashion. For example, by default, the text button notification template generates an “if” statement which only traps the

`TBUT_NFYHIT` notification. However, if it was more appropriate, the string resource which defines this notification could be altered such that it would generate a “case” construct instead.

Customizing the C Templates

In order to add interface functionality to the C templates, calls must be made to the Open Interface API. These procedures are used to manipulate the interface: perhaps to open a new window, to pass data from a list box cell to a text edit, or to expand the child nodes in a browser widget.

In Code Example 5, two lines of code have been added to the notification procedure generated by Open Editor to trap the mouseclick event for the button `QuitButton`. These two calls to the Open Interface API display the confirmation

```

static void Mod_QuitButtonNfy (TButPtr tbut, TButNfyEnum code)
{
    /* Only trap the "Hit" event */
    if (code != TBUT_NFYHIT) { TBUT_DefNfy(tbut, code); }

    /* Bring up an Ok/Cancel dialog defined in Genwin, the generic
       windows library, display the message contained in the string
       resource Mod.ConfirmStr and check for the boolean return code */
    if (ALRT_MsgOkCancel("Mod", "ConfirmStr") == BOOL_TRUE)
        EVENT_MainExit(); /* If Ok was pressed then exit */
}

```

Code Example 5: A customized text button notification routine.

message "Do you want to quit?" in a modal dialog box and check its result before terminating the application.

To further explore the details of the Open Interface API, refer to "Part 5 • The Widgets and their API".

Compiling & Linking an Open Interface Application

To complete the application, the C source files must be compiled and linked with the Open Interface libraries and those of the underlying windowing and operating system. Open Editor generates makefiles to simplify this process.

- On a Unix workstation the compiling and linking can be done with any standard Unix C compiler (ANSI or non-ANSI). On Unix platforms the X Windows library (libX11) is required to implement the low-level graphics functionality. On a Unix platform, the link command need to create the application win is:

```
cc -o win win.o /OpenInt/lib/app.o
-L/usr/OpenInt/lib -ldgw -ldtkit
-lndvgm -lndres -lndcore -lX11 -lm
```

- On a PC running Microsoft Windows, in addition to a Windows-compatible compiler and linker, the Windows library and a number of Windows header files are needed. So, if the Microsoft C compiler and linker (version 6.0 or later) are used, the Windows Software Development Kit™ will also be required.

- On a Macintosh, either Apple's Macintosh Programmer's Workshop™ (MPW v3.1 or later) or Symantec's THINK-C™ (v4.0 or later) can be used to compile and link an Open Interface application.

- On a VMS platform the standard VAXC compiler can be used. The DEC X Windows library DECW\$XLIBSHR is used to implement the low-level graphics functionality.

Finally, to create the binary resource file (.dat) from the text resource files (.rc), the Open Interface Resource Compiler, Rescomp, is used. On a PC this step would be done using the command:

```
C:\OPENINT\BIN\RESCOMP WINLIB.RC
C:\OPENINT\BIN\RESCOMP WIN.RC
```

Part 4 • Specifying the Look & Feel

Setting the Look & Feel Environment Variable

Before running a completed Open Interface application, the desired look and feel must be specified in the environment variable `OIT_LOOK`.

Under Unix this would be done as:

```
setenv OIT_LOOK MOTIF
```

or under DOS with

```
set OIT_LOOK=MSWINDOWS
```

On a Macintosh, the look and feel is set in an Open Interface preferences file.

Open Interface tailors the appearance and behavior of each widget according to the currently

specified look and feel. For example, the OPEN LOOK style guide dictates that a text button must have rounded corners. So, when `OIT_LOOK` is set to OPEN LOOK, the frame of all text buttons is drawn using the graphic primitive `DRAW_RoundRect()`. If instead `OIT_LOOK` is set to Motif, whose style guide specifies that text buttons must have square corners, the frame of all text buttons would be drawn using the function `DRAW_Rect()`. The appearance and behavior of each widget in the Open Interface toolkit are implemented in a similar fashion for the five look and feels. Figure 6 shows the same window under all five look and feels.

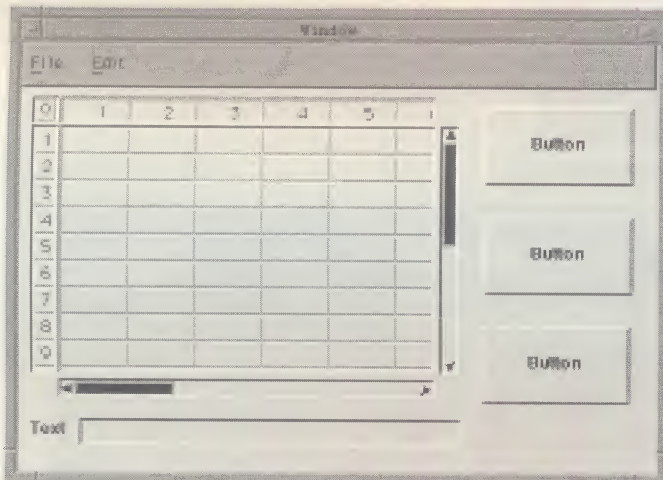


Figure 6a: Motif

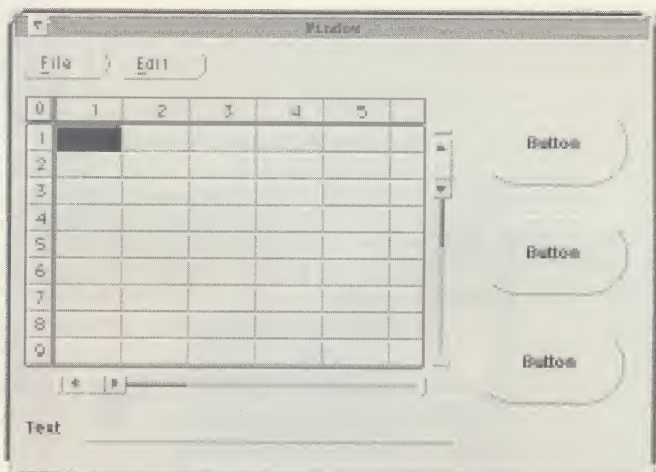


Figure 6b: OPEN LOOK

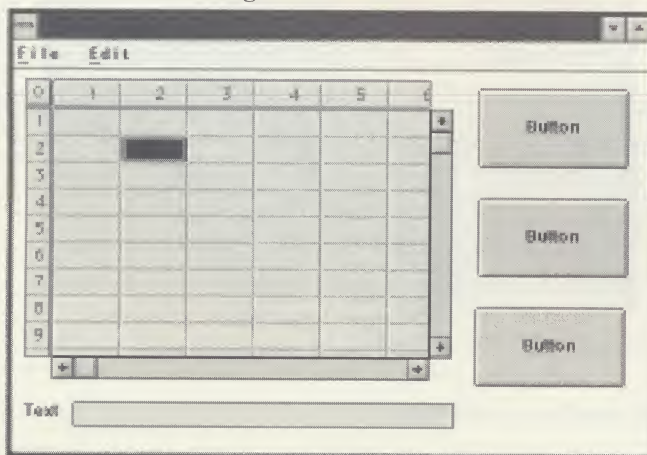


Figure 6c: Microsoft Windows

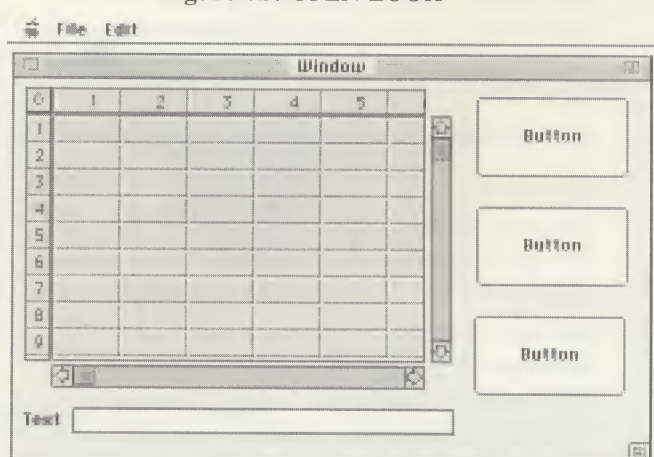


Figure 6d: Macintosh - System 7.0



Figure 6e: Presentation Manager

Porting Applications

To port an application to another windowing and operating system, all of the text resource files (.rc) and C source files that compose the application must be moved to the target platform, as shown in Figure 7. The C source files are then recompiled and linked with the Open Interface libraries and windowing system libraries for that particular platform.

The resource compiler, Rescomp, is then used to convert the text resource files into a machine-dependent binary resource file (.dat) for the new platform. Finally, the environment variable OIT_LOOK is set to specify the desired look and feel for the ported application.

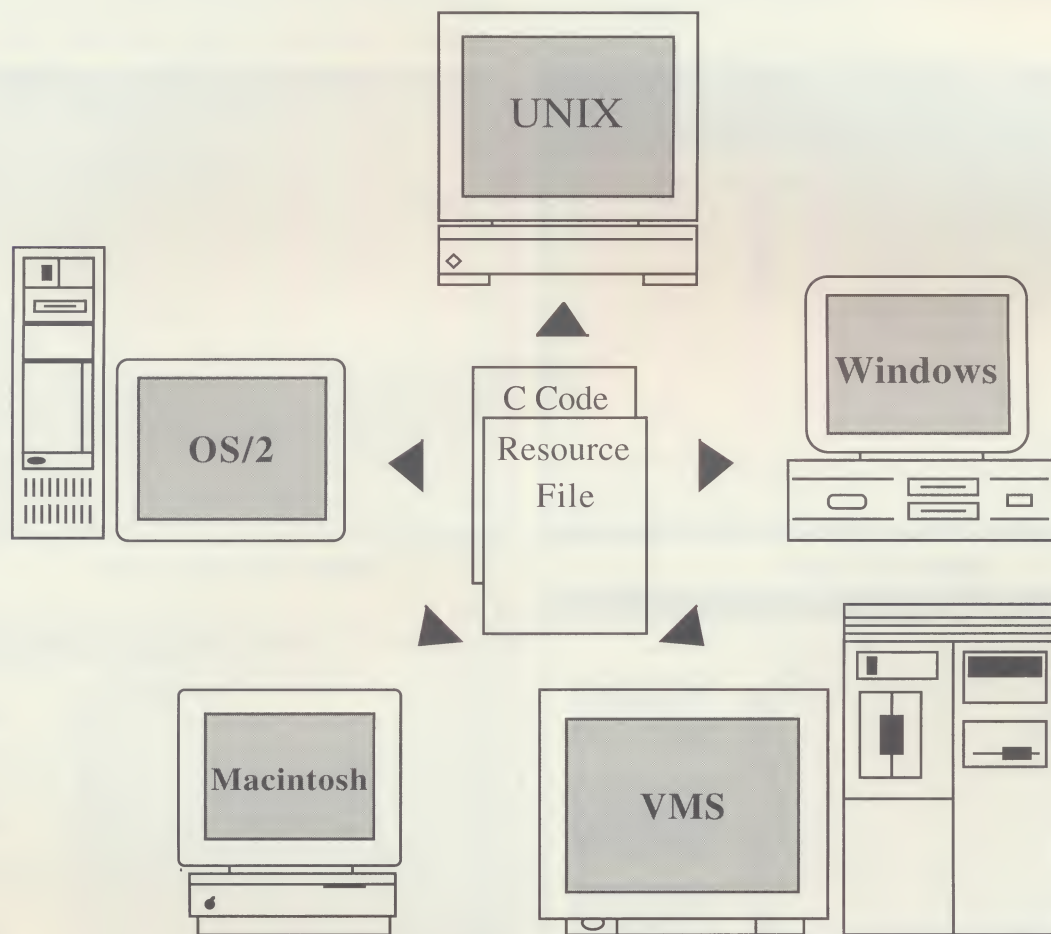


Figure 7: Porting an Open Interface Application

Part 5 • The Widgets & their API

One of the most powerful features of Open Interface is the level of abstraction of the API. This section describes the details of programming with Open Interface and offers examples which demonstrate how easy it is to implement rich interface functionality with just a few function calls. A selection of Open Interface widgets and their related API calls are examined in some detail.

Example 1: The Text Edit

The Open Interface text edit widget can be used as either a single- or multi-line input field, or as an output area for messages. It supports multiple, proportionally-spaced fonts; left, center, right and perfect justification; vertical and horizontal scrollbars; and will auto-scroll the buffer as the cursor leaves the visible area. It has a number of built-in selection behaviors, including those which select an entire word or line at a time.

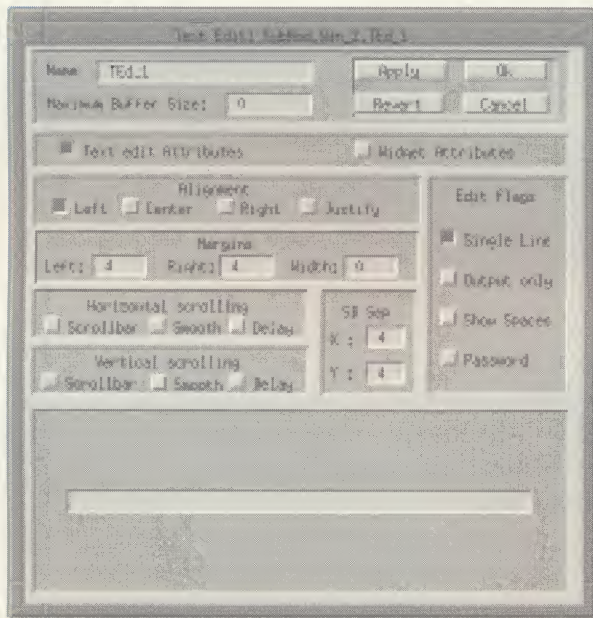


Figure 8: The Editor for the Text Edit Widget

Here is a synopsis of some key text edit APIs:

`TED_InsertStr()`

Inserts a string at the current insertion position. If a range of characters was selected, the new

string replaces the selected range.

`TED_InsertStrf()`

Same as `TED_InsertStr()`, but it accepts a format string and a variable list of arguments, just like the C library function `sprintf()`.

`TED_GetStr()`

Returns a pointer to the internal buffer of the text edit.

`TED_GetStrLen()`

Returns the number of characters in the buffer.

`TED_QueryStr()`

Copies the contents of the text edit into a buffer. The third argument specifies the size of the buffer. The routine will not copy more than `len-1` characters, and will terminate the buffer with a null character.

`TED_ClearAll()`

Clears the contents of the text edit.

`TED_SetInsertPos()`

Sets the insertion point at the index argument.

`TED_QuerySelRange()`

Retrieves the currently selected range.

`TED_SetSelRange()`

Changes the selection range.

`TED_SelectNone()`

Unselects any previously selected range and places the insertion point at the end of the buffer.

Example 2: The Table/List Box

The list box widget is an extremely powerful widget for displaying and manipulating data organized in rows and columns. In its simplest form it is a single column, multi-row, list box used to select a single item. In a more sophisticated instance it can implement a very large multi-row, multi-column spreadsheet with: horizontal and vertical scrolling, the ability to se-

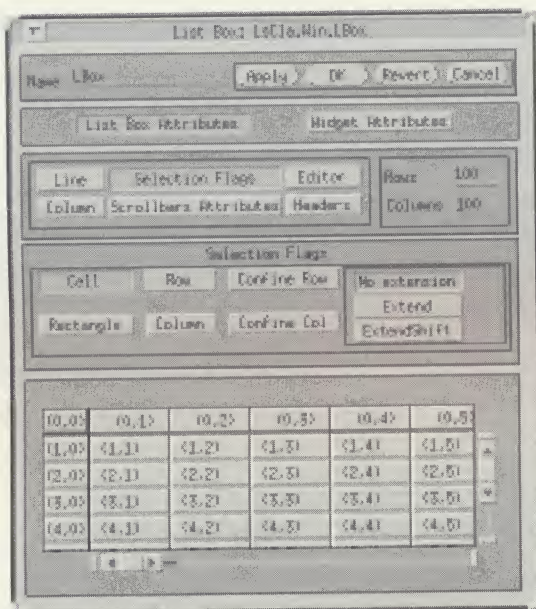


Figure 9: The Editor for the List Box Widget.

lect columns, rows or rectangles, row and column headings, dynamically resizable cells, edit-in-place features, and row and column insertion.

Furthermore, list box cells are not limited to displaying text — Open Interface drawing primitives can be used to draw graphics or icons within a cell.

In terms of memory usage, the Open Interface list box is very efficient. Memory is only allocated for the individual cells that are actually being used, empty cells within a populated region do not use any memory. This means that although a table could be very large, up to 32K rows by 32K columns, if only nine cells were being used, it would only use as much memory as a completely filled three-by-three table.

The list box API uses the concept of a “current cell”, i.e., a notification (`LBOX_NFYCELLDRAW`) is triggered for each cell which needs to be redrawn. Calls can be made to query the list box about its current state: to get the associated client data of the current cell, its selection state or its coordinates, and also to modify the position of the current cell. This simplifies the API

16

```
/* A static array holding list box data */
static char *ListBoxInfo[] = {
    "Cell(1,1)", "Cell(1,2)", "Cell(1,3)", "Cell(2,1)", "Cell(2,2)", "Cell(2,3)",
    "Cell(3,1)", "Cell(3,2)", "Cell(3,3)", "Cell(4,1)", "Cell(4,2)", "Cell(4,3)"
};

#define S_NUMROWS 4
#define S_NUMCOLS 3

static void S_FillLBox(WinPtr win)
{
    int row, col, data = 0;

    /* Get a pointer to the list box from its index */
    LBoxPtr lbox = (LBoxPtr)WIN_IndexToWgt(win, LBOX_INDEX);

    /* Move the current cell through each cell in the list box,*/
    /* and associate to it data from the array ListBoxInfo */
    for (row = 1; row <= S_NUMROWS; row++) {
        for (col = 1; col <= S_NUMCOLS; col++) {
            LBOX_GoColRow(lbox, col, row);
            LBOX_CurSetClientData(lbox, (ClientPtr)ListBoxInfo[data++]);
        }
    }
}
```

Code Example 6: List Box initialization routine.

significantly and is consistent with the object-oriented model of widgets as self-contained "state machines".

Code Example 6 shows the few lines of code needed to fill a multi-row list box with data from an array of character strings:

The routine shown in Code Example 7 responds to two list box notifications. In response to a draw-cell notification, `LBOX_NFYCELLDRAW`, it displays the text of the data associated to the cell. In response to a cell-click notification, `LBOX_NFYCELLCLICK`, it displays the data from

the selected list box cell in a text edit region in the same window. Note that because each widget has a pointer to its parent window, and that each window has a pointer to all of its child widgets, passing data among widgets is very easy.

Figure 10 shows the resulting application, with `OIT_LOOK` set to `OPENLOOK`. The standard main event loop and window initialization code (which calls `S_FillLBox()` before opening the window) have been added.

```
static void LBoxNfy (LBoxPtr lbox, LBoxNfyEnum code)
{
    WinPtr win;
    TEdPtr ted;

    /* Get a pointer to the text of the current cell */
    char *CellInfo = (char *)LBOX_CurGetClientData(lbox);

    /* If any text is attached to the current cell then draw it.
       Otherwise call the default list box notification procedure */
    switch (code) {
        case LBOX_NFYCELLDRAW:
            if (CellInfo != NULL) LBOX_DrawStr(lbox, CellInfo);
            else LBOX_DefNfy(lbox, code);
            break;

        /* If a click occurred on a filled cell, display its contents
           in the text edit region of the same window. To do this, first get
           a pointer to the parent window and then use this to get a pointer
           to its text edit region */
        case LBOX_NFYCELLCLICK:
            if (CellInfo != NULL) {
                WinPtr win = WGT_OF(lbox)->Win;
                TEdPtr ted = (TEdPtr)WIN_IndexToWgt(win, TED_INDEX);
                TED_SetStr(ted, CellInfo);
            }

        /* For any other notification, use the list box default notification routine */
        default:
            LBOX_DefNfy(lbox, code);
    }
}
```

Code Example 7: List Box notification routine.

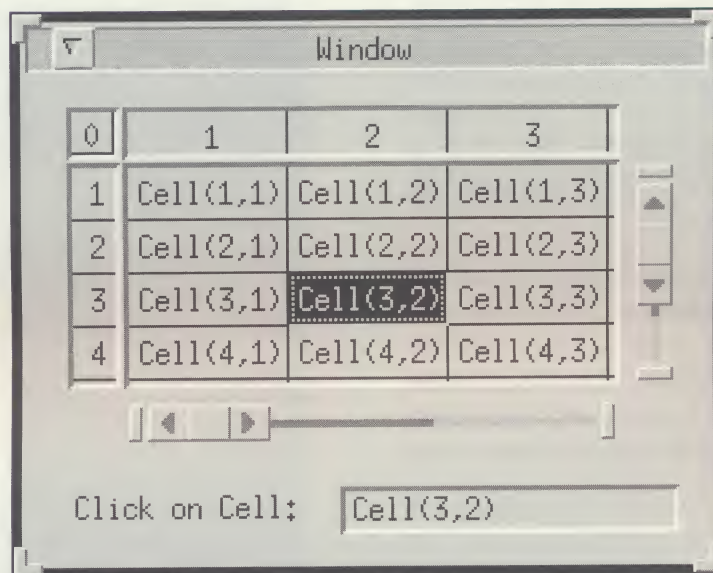


Figure 10: A sample Open Interface application using a list box and text edit.

A synopsis of some key list box APIs:

`LBOX_CurSetClientData()`

Associates data to the current cell. The client data can be any data type or structure.

`LBOX_CurGetClientData()`

Retrieves the client information attached to the current cell. If no information has been attached to it, NULL is returned.

`LBOX_GoColRow()`

Moves the current position to the cell specified by the row and column arguments.

`LBOX_GoUp/Down/Left/Right()`

Moves the current cell one cell in the specified direction.

`LBOX_GoAllocUp/Down/Left/Right()`

Moves in the specified direction to the next cell which has some information attached to it.

`LBOX_CurQueryCell()`

Returns the row and col position of the current cell.

`LBOX_IsCurSelected()`

Returns whether the current cell is selected or not.

`LBOX_RangeSelect()`

Selects/unselects the range specified

`LBOX_IsCurInUsedRange()`

Returns whether or not the current cell is in the range of cells to which information has been attached.

`LBOX_CurSetSelect()`

Sets the selection state of the cell according to a boolean argument.

`LBOX_InsertRow/Col()`

Inserts a row/col before the current cell. If the current cell is (1, 1), it will insert a new row/col at the top/left of the table. This call leaves the current cell on the newly inserted row.

`LBOX_DeleteRow/Col()`

Deletes the row or column containing the current cell. This call leaves the current cell at the same place.

`LBOX_QueryUsedRange()`

Returns the last row and column in the range of cells to which information has been attached.

`LBOX_SetRow/ColWidth()`

Sets the height/width (in pixels) of the column containing the current cell.

Example 3: The Browser Widget

The browser is a widget used to interactively display and navigate a hierarchy of objects. More specifically, it could be used to browse through a directory/file structure, a class/object

shows a compressed view of all of the nodes in the hierarchy and implements a rectangular view used to quickly pan around the hierarchy.

The browser API is similar in concept to that of the list box; again there is the concept of the "current node". Calls can be made to query the

```
/* Example node data structure */
typedef struct _NodeRec *NodeRecPtr;
typedef struct _NodeRec {
    char*   NodeLabel;           /* Text of current node */
    NodeRecPtr ChildNode;       /* Pointer to child record */
} NodeRec;

/* Node data from bottom to top. Three levels are used in this example */
static NodeRec Level3[] = {"GrandChild1", NULL}, {"GrandChild2", NULL},
    {"GrandChild3", NULL}, {NULL, NULL}};
static NodeRec Level2[] = {"Child1", Level3}, {"Child2", NULL}, {NULL, NULL}};
static NodeRec Level1 = {"Parent", Level2};
```

Code Example 8: Sample data structure for a browser widget.

hierarchy or a database/table/field hierarchy. As an example, the browser widget is used to display the Open Editor resource hierarchy.

Given a hierarchy of parent and child nodes, the browser widget will intelligently lay them out with the defined spacing and orientation and will draw all of the connecting lines between them. Nodes can be created and displayed dynamically, allowing users to selectively expand only pertinent branches of a hierarchy. Also provided is a browser overview widget which

status of the current node and to get its client data, or to navigate up, down or sideways relative to the current node. Like the list box, nodes are not limited to text, icons or graphics can also be drawn in a node.

Code Example 8 shows the data structure and data needed to initialize a network browser. Code Example 9 shows part of the window initialization procedure, which links the browser to its overview and establishes the parent node of the hierarchy.

```
BrowsPtr browser;
BOverPtr browsover;

/* Get a pointer to both widgets and link them together */
browser = (BrowsPtr)WIN_IndexToWgt(win, BROWSER);
browsover = (BOverPtr)WIN_IndexToWgt(win, BROWSOVER);
SOVER_LinkSArea((SOverPtr)browsover, (SAreaPtr)browser);

/* Create parent, associate it with Level 1 data and force a display */
BROWS_NewRoot(browser, VERT_DOWN);
BROWS_CurSetClientData(browser, (ClientPtr)&Level1);
BROWS_CurLayout(browser, HORZ_RIGHT);
```

Code Example 9: Initialization of the browser widget.

Code Example 10 shows the browser notification procedure. For any click on a node, the browser

will expand right to display its child nodes.

```
static void BrowserNfy (BrowsPtr browser, BrowsNfyEnum code)
{
/* Get a pointer to the current node */
NodePtr      Node = BROWS_CurGetNode(browser);

/* Get a pointer to the current node's associated data */
NodeRecPtr   NodeInfo = (NodeRecPtr)BROWS_CurGetClientData(browser);
NodeRecPtr   ChildInfo;

/* For a redraw notification, simply draw the text associated to a node */
switch (code) {
case BROWS_NFYNODEDRAW:
    BROWS_DefNodeDraw(browser, NodeInfo->NodeLabel);
    break;

/* Expand right when a node click occurs. First, make sure that the current
node has a child node defined in its data structure. Then make sure that
its children haven't already been displayed by trying to move the current
node down one generation. If the child nodes haven't yet been expanded,
BROWS_GoChildDown() will set the current node to NULL.*/
case BROWS_NFYNODECLICK:
    if (NodeInfo->ChildNode == NULL) break;
    BROWS_GoChildDown(browser, HORZ_RIGHT);
    if (BROWS_CurGetNode(browser) != NULL) break;

/* Create child nodes and attach data until no more children are found.
Since the previous call to BROWS_GoChildDown() will have left the current
node in the wrong place, and since each call to BROWS_NewChildRight()
also leaves the current cell on the new child node, BROWS_GoNode() must
be called each time to move the current node back to the correct node */
    for (ChildInfo=NodeInfo->ChildNode;ChildInfo->NodeLabel;ChildInfo++){
        BROWS_GoNode(browser, Node);
        BROWS_NewChildRight(browser);
        BROWS_CurSetClientData(browser, ChildInfo);
    }

/* Go back to parent node after the last call to BROWS_NewChildRight(),
recalculate the browser geometry and force a display */
    BROWS_GoNode(browser, Node);
    BROWS_CurLayout(browser, HORZ_RIGHT);

/* For any other notification, use the browser default notification.*/
    default:
        BROWS_DefNfy(browser, code);
    }
}
```

Code Example 10: The browser notification procedure.

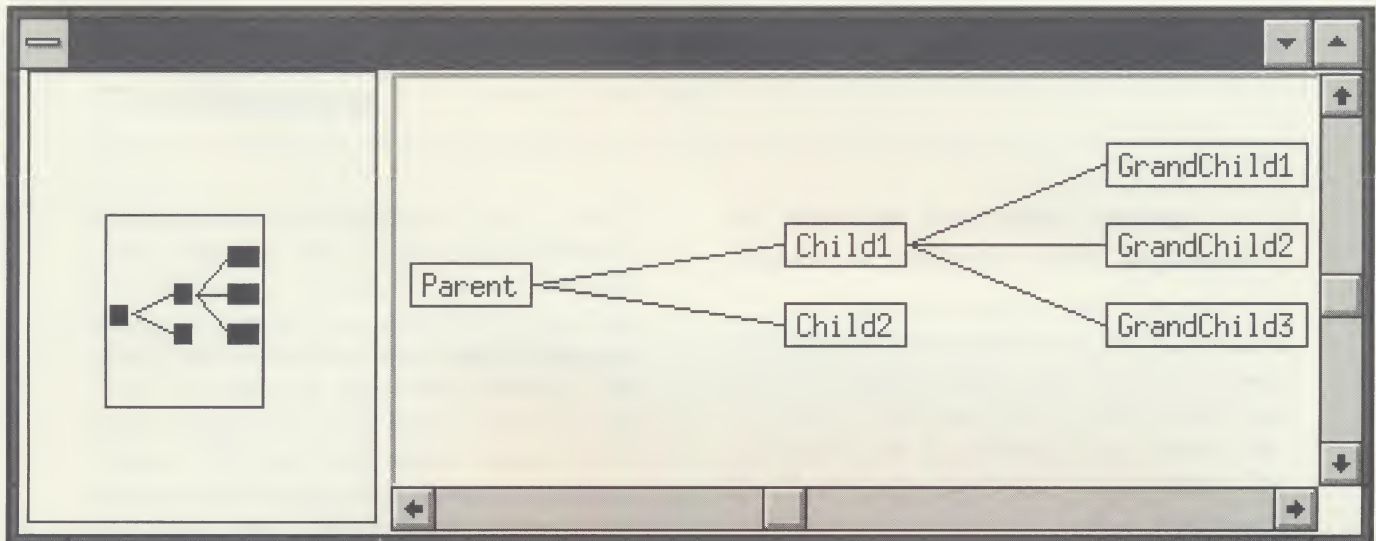


Figure 11: An Open Interface application using a browser and an overview.

Figure 11 shows the completed application. Here is a small sample of some other calls available in the Browser API:

`BROWS_CurQueryRects()`

Returns the coordinates of the current node. It is used when drawing icons or graphics inside a node. The first argument gets the coordinates of the node as if it were entirely visible on the screen and is used to compute drawing operation coordinates. The second argument gets the coordinates of the visible part of the node and is used to set the clipping region.

`BROWS_GoChildDown()`

Moves the current node down to its child node. If the current node does not have any child nodes, the current node becomes NULL.

`BROWS_Del()`

Deletes the current node and all its children. The current node is set to NULL after this call.

Menus and other Widgets in the Open Interface Toolkit

The examples above only describe three of the many widgets available in the Open Interface toolkit. There are many others including

menus, pop-ups, text areas, scrollable areas, icon buttons and bitmaps. Figure 12 shows an example of Open Interface cascading menu widget. "Part 7 • Extensibility and Custom Widgets" describes how to create and add new, custom widgets to the Open Interface toolkit.

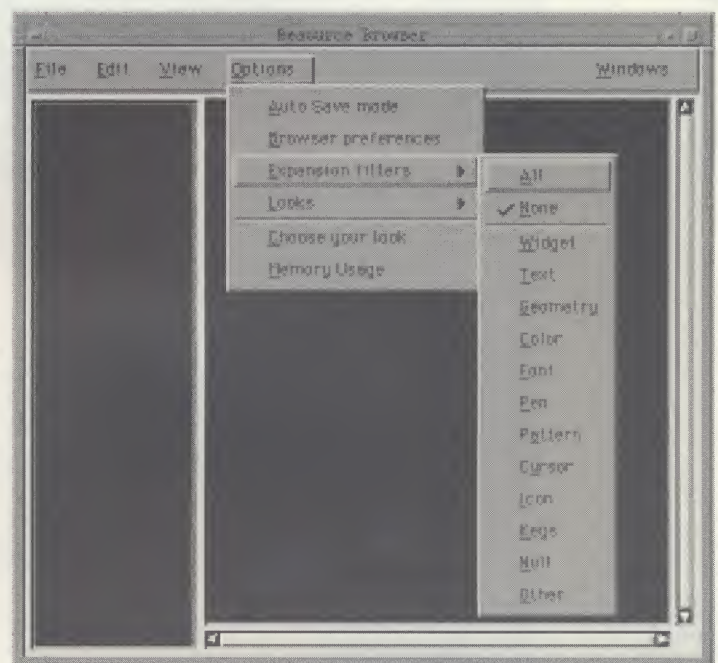
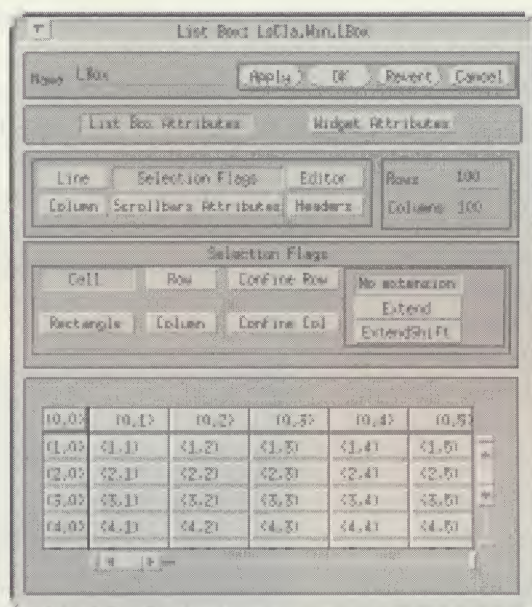


Figure 12: An example of the Cascading Menu Widget.



Part 6 • Integrating Open Interface & Databases

The portability and sophisticated widget set of Open Interface make it the perfect database front-end tool. Graphical database front-ends can be developed once and ported to any supported platform to give users identical database access regardless of their client platform. The Open Interface list box, browser and text edit widgets offer powerful visual metaphors for querying and navigating through databases, while their clean APIs make for straightforward and rapid front-end development.

A Simple Example Using Open Interface and Sybase™

The example below shows how Open Interface and Sybase's DB-Library™ can be integrated to provide database access with a minimal amount of coding. Any query can be entered in the text edit region and, by clicking on the "Query" button, be sent to the database for processing. The results are converted into character strings and displayed in the list box. Any number of result columns can be displayed.

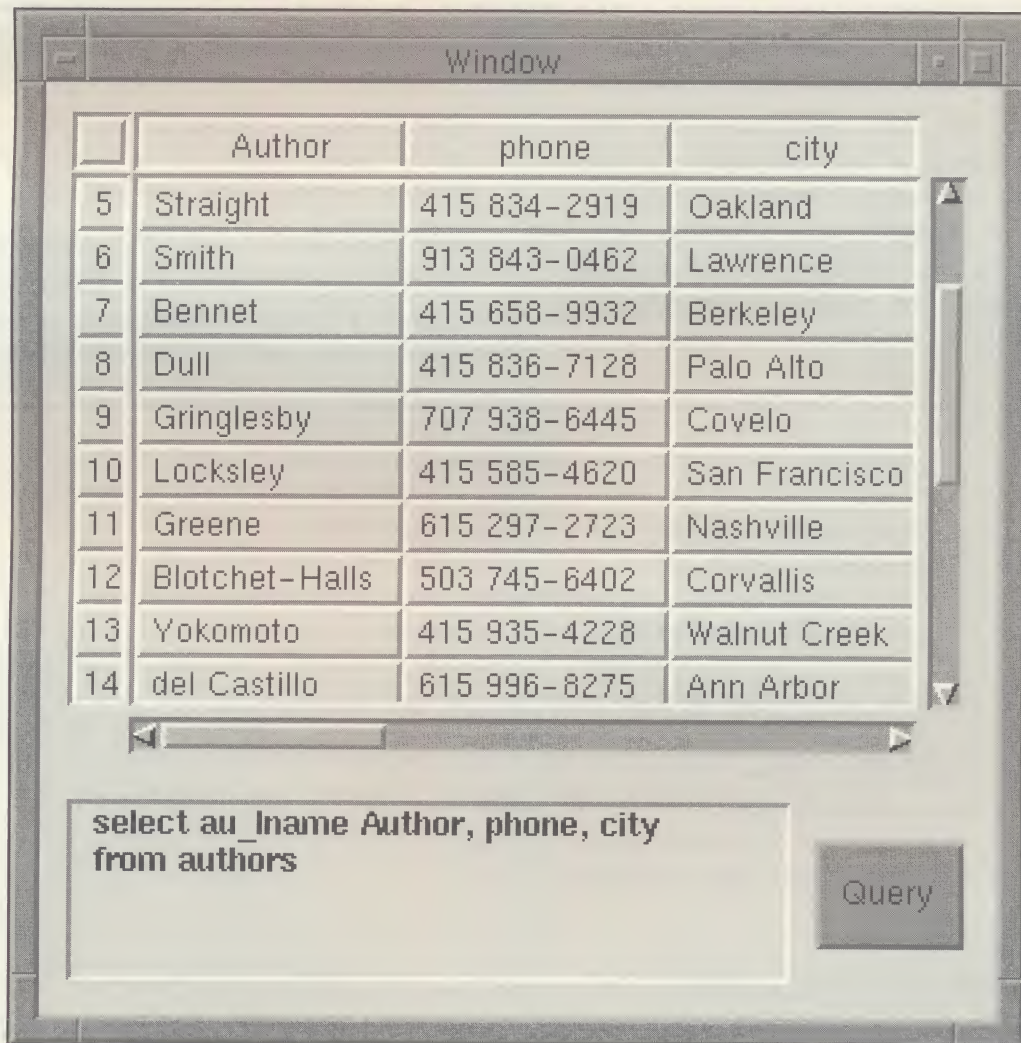


Figure 13: A simple Open Interface database front-end application.


```

/* Data structure used for all database communication */
DBPROCESS *dbproc;

S_DBLogin()
{
    LOGINREC *login;

    /* Initialize the database */
    if (dbinit() == FAIL)
        exit(ERREXIT);

    /* Allocate and fill a database login record */
    login = dblogin();
    DBSETLUSER(login, "oit");
    DBSETLPWD(login, "oit");

    /* Open a connection to the database*/
    dbproc = dbopen(login, NULL);
}

```

Code Example 11: The initialization and login routine.

S_DBLogin(), the initialization and login procedure shown in Code Example 11, is called just before the window is opened in the main() routine. It initializes some structures, fills a login record and establishes a connection with the database. The window is then initialized and opened.

Code Example 12 shows the notification routine for the "Query" button. Each time it is pressed, the query entered in the text edit region is sent to the database for processing.

The routine S_DBQuery(), shown in Code Example 13, doesn't actually associate any data to the list box cells; i.e., a client pointer is not

```

/* Text button notification -- triggers query on hit event */
static void SybMod_QueryButNfy(TButPtr tbut, TButNfyEnum code)
{
    /* Get a pointer to the text edit and list box widgets */
    WinPtr win = WGT_OF(tbut)->Win;
    LBoxPtr lbox = (LBoxPtr)WIN_IndexToWgt(win, LBOX);
    TEPtr ted = (TEPtr)WIN_IndexToWgt(win, QUERYTED);

    /* When the button is clicked, pass the query to the database,
       process the results, and then send a notification to the list
       box to get it to redraw itself.*/
    if (code == TBUT_NFYHIT)
    {
        S_DBQuery(lbox, TED_GetStr(ted));
        WGT_SendNfy(lbox, LBOX_NFYREDRAW);
    }
    else
        TBUT_DefNfy(tbut, code);
}

```

Code Example 12: The Query button notification routine.

being set for each cell in the list box. Instead, each time the cell redraw notification is received, its value is looked up by row and column in the result buffer. The DB-Library function `dbsetopt()`, called just before the query is processed, is used to allocate this result buffer. Once the query has been executed successfully, `dbnextrow()` is called repeatedly to read all of the rows into the result buffer. However, in order to get the list box scroll bars to display correctly, the list box needs to know how many rows and columns of data will be displayed. To do this, the current cell is moved to the cell which corresponds to the last column of the last row in the database result buffer and some

dummy data is associated to it.

The notification routine for the list box is shown in Code Example 14. `S_LBoxDraw()` looks up and displays the value of the current cell. First, the row and column values of the current cell are obtained. These values are tested to ensure that the cell falls within the boundaries of the result buffer. If the current cell is in row zero, the result column headers are displayed. The routine `dbconvert()` converts the result from any data type to a character value and copies it into the buffer `chardata[]`. Finally, the result is displayed as a text string in the current cell.

```
S_DBQuery(LBoxPtr lbox, char *query)
{
    /* Pass the query to the query buffer */
    dbcmd(dbproc, query) ;

    /* Allocate 100 rows in the result buffer */
    dbsetopt(dbproc, DBBUFFER, "100") ;

    /* Process the query */
    dbsqlxec(dbproc) ;

    /* Set up query results */
    if (dbresults(dbproc) == SUCCEEDED)

    /* Read all of the rows into the command buffer */
    { while (dbnextrow(dbproc) != NO_MORE_ROWS) ;

    /* In order to get the list box scroll bars to appear
       correctly, the boundaries of the valid region must be defined
       by allocating some dummy data to the bottom-right cell.
    */
        LBOX_GoColRow(lbox, dbnumcols(dbproc), DBLASTROW(dbproc)) ;
        LBOX_CurSetClientData(lbox, "Dummy") ;
        LBOX_GoHome(lbox) ;                               /* Reset the current cell */
    }
    else
        ALRT_Ok("Query failed\n") ;
}
```

25

Code Example 13: The database query routine.

With the standard window initialization and main routines included, the code is compiled

and linked with the Open Interface and Sybase libraries to complete the application.

```

/* List box notification - traps cell redraw notifications */
static void SybMod_LBoxNfy(LBoxPtr lbox, LBoxNfyEnum code)
{
    switch (code) {
        case LBOX_NFYCELLDRAW:
            S_LBoxDraw(lbox) ;
            break ;
        default:
            LBOX_DefNfy(lbox, code);
    }
}

S_LBoxDraw(LBoxPtr lbox)
{
    int    LBoxRow = LBOX_CurGetRow(lbox) ;
    int    LBoxCol = LBOX_CurGetCol(lbox) ;
    DBCHAR chardata[DBMAXCHAR] ;

    /* If the current cell is outside the boundaries of the results
       area, just call the default list box notification routine. */
    if ((DBLASTROW(dbproc)< 1)
        || (LBoxCol > dbnumcols(dbproc)) || (LBoxCol < 1)
        || (dbgetrow(dbproc, LBoxRow) == FAIL)
        || ((LBoxRow > 0) && (dbgetrow(dbproc, LBoxRow) == NO_MORE_ROWS)))
        LBOX_DefNfy(lbox, LBOX_NFYCELLDRAW);

    /* If the current cell is in the header region, display
       the database column headers */
    else if (LBoxRow < 1)
        LBOX_DrawStr(lbox, dbcolname(dbproc, LBoxCol)) ;

    /* Otherwise, look up the value of the current cell in the Sybase
       results buffer and convert it into a character string */
    else
    {
        dbconvert(dbproc,
            dbcoltype(dbproc, LBoxCol),          /* Determine data type */
            dbdata(dbproc, LBoxCol),              /* Pointer to data */
            dbdatlen(dbproc, LBoxCol),            /* Length of data */
            SYBCHAR, chardata, -1) ;              /* Copy into chardata */

        /* Display the result in the current cell */
        LBOX_DrawStr(lbox, chardata) ;
    }
}

```

Code Example 14: The list box notification routine and cell draw routine.

Porting the Database Front-end Application

Since both the Open Interface and Sybase libraries are available for Unix, Windows, OS/2 and Macintosh platforms, a completely portable application can be developed to offer identical functionality on all of these platforms without any code changes. For each platform, the source code is simply recompiled and relinked with the appropriate libraries for the target platform.

Database Access with Open Interface and NEXPERT Object

NEXPERT Object offers database connectivity to more than a dozen relational databases, including Ingres™, Informix™, Oracle™, Sybase™, Rdb™, VAX SQL™, SQL/DS™, DB2™ and to flat-file formats such as Lotus™, Excel™ and dBASE III™. In conjunc-

tion with Open Interface, NEXPERT can be used to access data from multiple sources and present it in a single user interface. Any SQL query can be maintained in a NEXPERT rule, and triggered through the NEXPERT API from an Open Interface notification routine. The query results, passed to the NEXPERT class and object structure, can be mapped into any Open Interface widget, such as a list box or a browser.

There are three steps involved in accessing a database from NEXPERT. First, an SQL query must be entered in the query field. Then the relationship between the results (or inputs) of the query and the NEXPERT class and object structure must be defined. Finally, the type of database being accessed (i.e., Oracle, Rdb, Excel SYLK format, etc.) must be specified. This flag can be easily modified to change the target database for any query. For example, an SQL query written for an Informix database can be switched to use Oracle simply by changing this one flag.

Begin

Query

End

Name

Cursor In

Database Type

- INGRES
- NXP
- NXPDB
- ORACLE**
- RDB

Database Fields	NEXPERT Properties
DB_MODEL	MyCar.Model
DB_MODEL_DATE	MyCar.Model_date
DB_PRICE	MyCar.Price
DB_SPORTIVE	MyCar.Sportive

☐ Create New Record

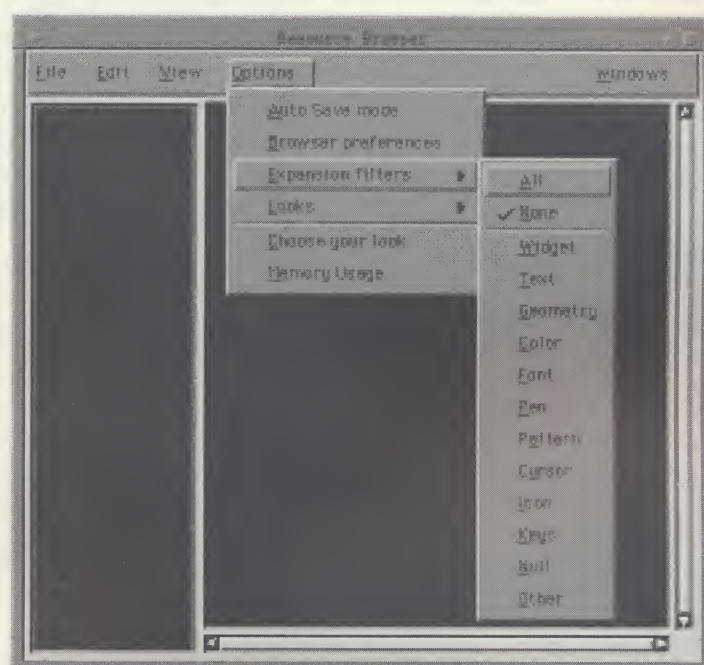
☐ Write Unknown

☐ New File

OK

Cancel

Figure 14: Specifying a SQL query with the NEXPERT Object.



Part 7 • Extensibility & Custom Widgets

The layered API model of Open Interface is at all times extensible and portable. The powerful widget functionality offered by the Open Interface toolkit can be extended to include new custom widgets, specific to a particular line of business or class of application. Because cus-

ground color, background color, font, pen, pattern, etc. Associated with those attributes are the functions which manipulate them, like `WGT_SetFgColor()` and `WGT_SetFont()`, for example.

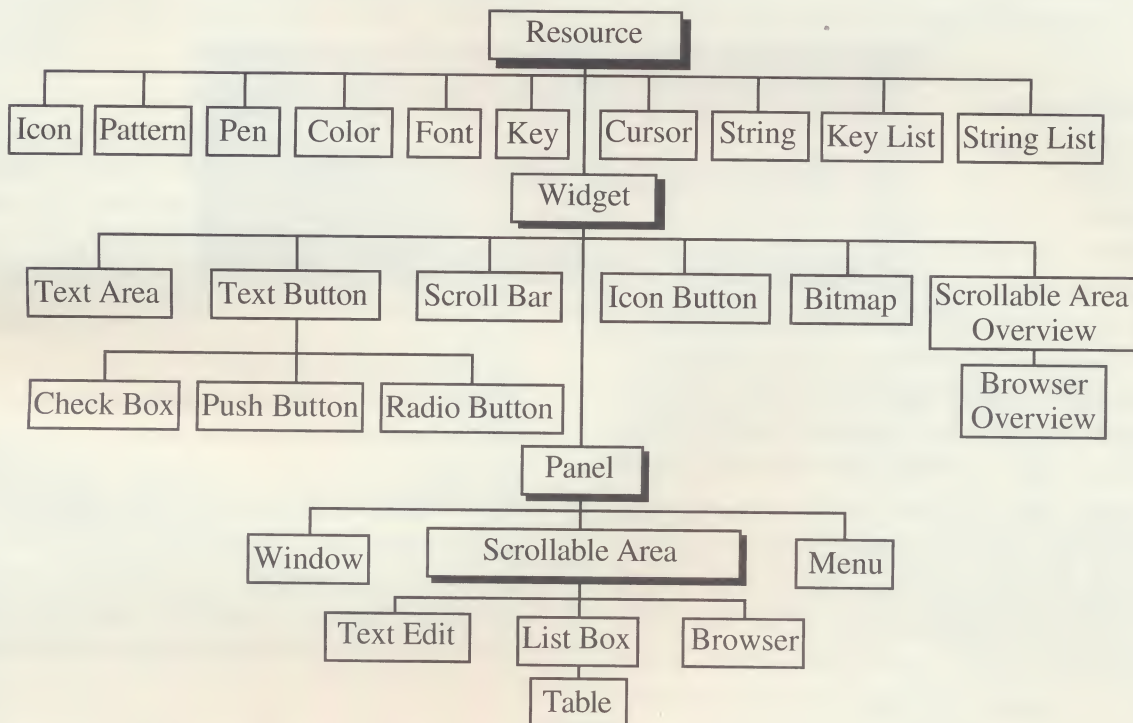


Figure 15: The Open Interface Widget Hierarchy.

tom widgets are described using calls to the Virtual Graphics Machine in response to a consistent set of events generated by Open Interface, they are always completely portable — they do not need to be re-implemented under each native windowing systems.

The implementation of new widgets is further simplified by Open Interface's object-oriented design; new widgets can sub-class from existing widget classes in order to inherit common attributes and procedures. For instance, by subclassing from the widget class, a structure is inherited containing those attributes which are common to all widgets: position, size, fore-

New widget classes can be seamlessly incorporated into the Open Interface development environment; hooks into Open Editor and Rescomp are available in order to register custom widgets. In addition, by adding string resources to the Open Editor resource library, template notification procedures for custom widget classes can be generated. Even custom widget editors can be linked into Open Editor.

The process of creating a new widget class consists of defining its data structure, developing a default notification procedure, and specifying the necessary initialization routines.

The Inheritance Mechanisms of Open Interface

The inheritance mechanisms of Open Interface make defining the data structure of a new widget class a straightforward procedure. In this example, the slider widget shown in Figure 16 is created. It is a very simple widget, which given a minimum and maximum value, positions a square "thumb" at the current position.

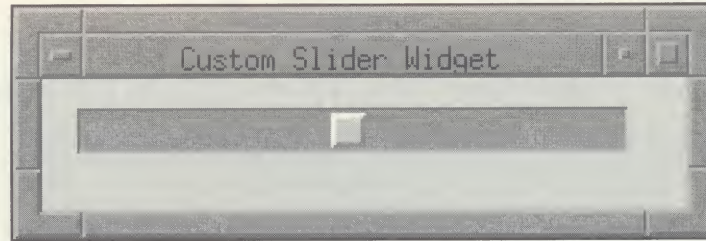


Figure 16: An example of a custom widget — the simple slider widget.

In order to create the data structure for this simple widget, those attributes common to all widgets can be inherited from the class "widget" via a C macro (WGT_REC), and then augmented with those fields which are specific to the example slider widget, such as its minimum, maximum, and current value. This

structure, shown in Code Example 15, would best be maintained in a header file, accordingly named `slidpub.h`.

The persistent fields of the new widget class, i.e., those which are to be maintained in resource files, must be declared in a structure, `S_Fields[]`, to the resource manager. For the slider widget, it makes sense to store the minimum and maximum values as persistent fields in the resource files. The macro

`PFLD_TYPEINT16` tells the resource manager that `MinVal` and `MaxVal` will be stored as integers; the macro `PFLD_CATSIZ` declares them to be "size" fields, as opposed to some other low-level resource like a color (`PFLD_CATCOLOR`) or pen (`PFLD_CATPEN`) field.

```
#include <wgtpub.h>

/* Data structure for a custom slider widget */
typedef struct _SliderRec {
    WGT_REC      /* Macro to inherit generic widget structure */
    int MinVal;   /* persistent field */
    int MaxVal;   /* persistent field */
    float CurVal; /* volatile field */
} SliderRec;

typedef struct _SliderRec *SliderPtr;          /* Pointer to the SliderRec struct */

/* Declare the persistent fields of the slider widget to the resource manager */
static PFLDDescRec S_Fields[] = {
    { "MinVal", C_OFFSET(SliderRec, MinVal), PFLD_TYPEINT16, PFLD_CATSIZ },
    { "MaxVal", C_OFFSET(SliderRec, MaxVal), PFLD_TYPEINT16, PFLD_CATSIZ },
    { (char*)0, 0, 0, 0, }
};
```

Code Example 15: The data structure and persistent fields of the simple slider widget.

The Drawing Primitive API

The bulk of the work involved in creating a custom widget class derives from the definition of its default notification procedure. This procedure will implement the widget's default behavior: specifying how it is drawn or resized and how it handles the various mouse and keyboard events. The display behavior of any custom widget is defined using calls to the drawing primitives of the Virtual Graphics Machine. Here are a few of the available routines.

`DRAW_SetColors()`

Sets the "on" (second arg) and "off" (third arg) colors. The on color defines what happens to the bits which are set in the drawing operation. The off color defines what happens to the bits which are cleared in the drawing operation.

`DRAW_SetPattern()`

Sets the pattern. If NULL is passed, the default pattern of the widget is used.

`DRAW_SetPen()`

Sets the pen. If NULL is passed, the default pen of the widget is used.

`DRAW_SetFont()`

Sets the font. If NULL is passed, the default font of the widget is used.

`DRAW_Line()`

Draws a line. The second argument is the origin of the line and the third argument the end of the line. Both the origin and the end points are drawn.

`DRAW_Rect()`

Draws a rectangular box. The frame is drawn with the current pen, the interior with the current pattern. If the pen is wider than 1 pixel, the inside of the rectangle is reduced but the outside of the frame does not change.

`DRAW_RoundRect()`

Draws a box with rounded corners in the rectangle passed as second argument. The third argument defines the x and y axes of the ellipses used to draw the corner.

`DRAW_Icon()`

Draws the icon passed as third parameter inside the rectangle passed as second argument.

`DRAW_Ellipse()`

Draws an ellipse bounded by the rectangle passed as the second argument.

`DRAW_Arc()`

Draws an elliptical arc. It is similar to `DRAW_Ellipse()` except that the third argument specifies where the arc starts and ends, as well as how the arc is terminated.

Code Example 16 shows the default notification routine for a simple slider widget. The first structure, `SliderNfyEnum`, uses a C macro to inherit the standard widget notifications (and to prefix them with `SLIDER` to create, for instance,, `SLIDER_NFYREDRAW`). Although the list of widget notifications is quite extensive, new, composite notifications, like a double-click, can also be defined. For this example widget, the default notification `SliderDefNfy()` will only trap the redraw and mouseclick events.

The function `S_Redraw()` defines how to draw the slider. The structure `box` holds the current size of the widget; `Ori.x` and `Ori.y` store the position of the top left-hand corner relative to its parent window (or panel). `Ext.x` and `Ext.y` hold the width and height of the widget.

First, the size and position of the square thumb, `CurRect`, is computed. For simplicity's sake, the slider is assumed to be a horizontal one. `Ext.x` and `Ext.y` are set to create a square thumb slightly shorter than the widget rectangle. The x origin is computed to center the thumb at the distance `CurVal` between `MinVal` and `MaxVal`, two pixels below the border of the widget.

Next the drawing context for the widget frame is set to use the current background color and a 3D hole pen. The widget frame is then drawn using a call to the Virtual Graphics Machine routine `DRAW_Rect()`. Finally, a similar procedure is used to set the drawing context and to draw the slider thumb.


```

#include <wgtpub.h>
#include <slidepub.h>

typedef enum {
    WGT_NFYINHERIT(SLIDER)          /* Inherit widget notifications */
} SliderNfyEnum;

/* Default Notification Routine */
void SLIDER_DefNfy(SliderPtr slider, SliderNfyEnum code)
{
    ERR_TRACEIN;
    switch (code) {
        case SLIDER_NFYREDRAW:
            S_Redraw(slider);          /* See code below */
            break;
        case SLIDER_NFYMOUSECLICK:
            S_Click(slider);
            break;
        default:
            WGT_DefNfy((WgtPtr)slider, (WgtNfyEnum)code);
    }
    ERR_TRACEOUT;
}

/* Drawing */
static void S_Redraw (SliderPtr slider)
{
    RectPtr box = &slider->Box;      /* Current widget boundaries */
    RectRec CurRect;
    int MinVal = slider->MinVal;
    int MaxVal = slider->MaxVal;
    float CurVal = slider->CurVal;

    /* Calculate position and size of square thumb slider from CurVal */
    CurRect.Ext.x = box->Ext.y - 4; /* Leave a 2 pixel gap from frame */
    CurRect.Ext.y = box->Ext.y - 4; /* on all sides */
    CurRect.Ori.x = (CurVal-MinVal)/(MaxVal-MinVal)*box->Ext.x - (CurRect.Ext.x/2);
    CurRect.Ori.y = 2;

    /* Draw widget rectangle using 3D pen and background color */
    DRAW_SetColors((WgtPtr)slider, slider->BgColor, slider->BgColor);
    DRAW_SetPen((WgtPtr)slider, PEN_Hole());
    DRAW_Rect((WgtPtr)slider->Panel, box);

    /* Draw thumb with 3D pen and foreground color */
    DRAW_SetColors((WgtPtr)slider, slider->FgColor, slider->FgColor);
    DRAW_SetPen((WgtPtr)slider, PEN_Bump2());
    DRAW_Rect((WgtPtr)slider, &CurRect);
}

```

Code Example 16: The default notification routine for the simple slider widget.

The Initialization Routines

A new widget class will also need some default, low-level resources. In this example the resource library `extkit.dat` has been created to hold extensions to the toolkit. The resource module for the slider, shown in Code Example 17, defines its class name (as it will appear in the list of widget in Open Editor) and its default color resources:

Code Example 18 shows `S_InitRes()`, which initializes the persistent fields of the slider widget to their default values. It calls `WGT_DefInitRes()` which sets the persistent generic widget fields to their default values. The remaining two lines of code set `MinVal` and `MaxVal` to 0 and 100, respectively. These values will be used as defaults whenever a new instance of the slider is dynamically created.

The initialization routines for the new widget class are also defined. They register the new slider widget class with the resource manager and load its required resources.

```
(RMod.Compile
    Name:      "Slider"
    Lib:       "ExTkit"
    FileName:  "slider.rc"
)
(String.Compile
    Name:      "Slider.ClassName"
    Text:      "Slider"
)
(Color.Compile
    Name:      "Slider.ColorFg"
    Flags:     0x1
    Red:       0
    Green:     0
    Blue:      0
)
(Color.Compile
    Name:      "Slider.ColorBg"
    Flags:     0x1
    Red:       55296
    Green:     55296
    Blue:      48896
)
```

Code Example 17: The slider resource module.

```
/* Resource initialization */
static void S_InitRes(SliderPtr slider)
{
    WGT_DefInitRes((WgtPtr)slider); /* Set widget def values */
    slider->MinVal = 0;              /* Default MinVal */
    slider->MaxVal = 100;            /* Default MaxVal */
}

static WgtClassRec S_Class = {
    sizeof(SliderRec), S_ModuleName, S_Fields,
    RES_CLASSFLAGWGT, (ResNfyProc)SLIDER_DefNfy, (ResProc)S_InitRes,
};

/* Initialization for the resource manager */
void SLIDER_InitRes()
{
    S_Class.ParentClass = WGT_Class();
    WGT_REGISTERCLASS(&S_Class);
}

/* Initialization routine for Open Editor and Rescomp */
void SLIDER_LibInit()
{
    SLIDER_InitRes();
    RLIB_LoadLibFile("ExTkit", "extkit.dat");
}
```

Code Example 18: The initialization routines for the simple slider widget.

Finally, Code Example 19 shows the template main routines of the Open Editor and Rescomp, modified to include the new widget class.

All of the above C code can be maintained in one source file, `slider.c`. These routines can be compiled and linked into new versions of Open Editor and Rescomp. The link step for recreating Open Editor on a Unix platform would be:

```
cc -o openedit openedit.o slider.o app.o
-L/usr/OpenInt/lib -lndgw -lndtkit
-lndvgm -lndres -lndcore -lX11 -lm
```

The slider widget will now be available from within the Window editor and can be painted on the screen in the same manner as any other widget in the toolkit. The Open Interface Resource Compiler, Rescomp, can be rebuilt such that it can read and write the persistent fields of the slider. In order to use the widget in an Open Interface application, a call to `SLIDER_LibInit()` must be added before `APP_InitLast()` in the `main()` procedure, and the makefile must include `slider.o` in the list of files to link.

```
/* Open Editor main entry point */
#include <appub.h>
#include <resedpub.h>
#include <slidepub.h>

int main(int argc, char** argv)
{
    APP_InitFirst();
    RESED_LibInit();
    SLIDER_LibInit();           /* Initialize new slider widget class */
    APP_InitLast();
    RESED_Main(argc, argv);
    RESED_LibExit();
    APP_Exit();
    return(1);
}

/* Rescomp main entry point */
#include <rcomppub.h>
#include <slidepub.h>
int main(int argc, char** argv)
{
    RCOMP_Init();
    SLIDER_LibInit();           /* Initialize new slider widget class */
    RCOMP_Main(argc, argv);
    RCOMP_Exit();
    exit(0);
}
```

Code Example 19: The Open Editor and Rescomp main routines — modified to initialize the custom slider widget class.

Part 8 • Questions & Answers

General

Can Open Interface application development be done on any platform ?

Yes. Since Open Editor is built entirely on the Open Interface libraries, it is completely portable and offers identical functionality on every platform. This means that there is no platform of choice for Open Interface development; an application can be developed on any of the supported platforms and ported to any other.

As the interface complexity of an application grows, what happens to the size of the executable?

Given Open Interface's object-oriented implementation, a substantial portion of the interface code can be reused. Thus, as an application's interface complexity grows significantly, the size of the executable needed to implement the interface will only increase marginally.

The executable size of an application can also be optimized by stubbing out unused widget classes. For instance, if the browser widget wasn't used in an application, about 100K of code can be saved by stubbing it out. This would be done by linking in void initialization and default notification routines before the Open Interface toolkit library.

In addition, wherever possible, the Open Interface libraries are implemented as shared images, either as DLLs (Dynamically Linked Libraries) on the PC, as shared libraries under Unix and as shareable images under VMS.

Is there a performance penalty when using Open Interface?

Compared to applications built with the Motif, OPEN LOOK and Xt Intrinsics toolkit libraries, the responsiveness of Open Interface applications is usually superior. This is due largely to the fact that, unlike Motif and OPEN LOOK widgets, an X Windows window is not opened for each Open Interface widget, and also that

Open Interface caches and shares its low-level resources (like fonts, colors, etc.) in a very efficient manner. This performance improvement is particularly noticeable on X-server terminals. Under the Macintosh, Windows and PM windowing systems, the responsiveness of applications built with Open Interface is always comparable to a similarly behaving native application.

Can calls to Open Interface be combined with calls to the native windowing system?

Yes. For instance, under X Windows, calls can be made to the Xlib API to do some very special graphics, or perhaps to re-use some existing, intricate Xlib code. This is also true for Windows, PM and the Macintosh. However, in all cases, whenever calls to the native windowing environment are made, portability is immediately lost.

What about porting existing applications under Open Interface so they can become portable?

Within the existing code, the calls which were made to a specific GUI API can be replaced with calls to the Open Interface API. Once these modifications have been made, the software will be portable across all platforms.

Can a GUI be modified without recompiling the application?

The attributes (resources) of any interface can be modified without any recompilation. The latter is only necessary if widgets are added to or removed from an application.

How are menus handled and ported across platforms?

Menu bars, popup-menus and cascading menus are completely handled by Open Interface and are drawn according to the current look on each platform. With Open Editor's menu editors menu bars, pop-up menus and cascading menus can be quickly and easily developed. Labels, icons, check-marks, horizontal or vertical separators

can be specified as attributes of any menu or menu item. Except the Macintosh, all look and feels attach the menu bar inside the window. On Macs using the native Mac look and feel, the standard menu bar is used.

What happens to the size of windows and widgets when I port from a machine with a high-resolution display to one with a low-resolution display?

Open Interface measures graphic resources in terms of absolute pixels. So, some large windows will have to be resized for delivery to a smaller (like PC EGA, VGA or 9" Mac screen). However, it is important to point out that this won't require any changes in the C source code, only the resource files.

What happens to fonts when I port applications?

Open Interface uses the font resources of the underlying platform. Using the font editor in Open Editor, Open Interface logical font resources are mapped to the local fonts of the underlying windowing system. When porting an application from one platform to another, Open Interface logical font names may have to be re-mapped to the most appropriate native font of the underlying windowing system. This could be done using Open Editor or simply by modifying the text resource file.

What happens to colors when I port applications?

Open Interface also uses the color resources of the underlying platform. With Open Editor's color editor RGB values can be mixed and assigned to an Open Interface logical color resource. Again, when porting an application from one platform to another, Open Interface logical color names may have to be re-mapped to the most appropriate color available on the underlying windowing system.

How does Open Interface work on monochrome screens?

On Unix, VMS and PC platforms, an environment variable `OIT_COLOROPT` can be set to `MONOCHROME`. This will use the standard NTSC

algorithms to optimally display the interface in black and white. On a Macintosh Open Interface applications will automatically switch to monochrome whenever a B&W display is used. Note that fill patterns can also be used in widgets and windows to improve the clarity of the interface on monochrome screens.

Are the C templates generated by Open Interface portable?

Yes. Open Interface can generate ANSI C templates on every platform. These templates are fully portable across platforms and will compile and link with the local compiler and linker.

Can incremental development be done with Open Interface?

Yes. Since the templates generated by Open Editor will compile without any modification, functionality can be added to the application on a widget-by-widget basis. If a notification procedure has not been customized for a particular instance of a widget, its default notification routine will be called to implement its default behavior.

Are Open Interface applications compatible with applications built with the native toolkits?

All applications developed with Open Interface co-exist with applications written using the native windowing systems and toolkits. Open Interface applications behave as good citizens under all of the window managers, without any modification.

Is Cut and Paste with non-Open Interface applications supported?

Yes. Text cut and paste is directly supported by Open Interface, from application to application or window to window, using the normally associated keyboard bindings.

How is the cursor shape mapped to external events?

Cursors are managed by Open Interface. A cursor can be specified as a window, panel or widget resource (meaning that the cursor will auto-

matically change to the selected cursor as it passes over the window or widget), or can be remapped at any time under program control.

Can I trap non-graphic events? For instance, what if I wanted to display data from a real-time feed?

Yes, this can be done using the Unix `signal()` or `select()` functions. An example program of how to do this under Unix is provided with the Open Interface kit.

Localization & Multi-byte Character Support

Does Open Interface support the Kanji character set? How can an Open Interface application be localized?

Open Interface has true multi-byte character provisions for Kanji and full Native Language Support. The localization of an application re-

quires only that the string messages, labels, and menu choices be translated in the resource file, and that some widgets like text buttons and menus be resized to accommodate the new strings. All of this can be done within Open Editor — the application code would not even need to be recompiled. In this way, a single executable can be maintained with a different binary resource file for each supported language. All of the Open Interface string manipulation functions support multi-byte character strings — thus, the creation of Kanji applications will not require any changes to the underlying source code.

Microsoft Windows

Are the Microsoft Windows and Presentation Manager Open Interface libraries Dynamically Linked Libraries (DLLs)?

Yes. All six of the Open Interface libraries are

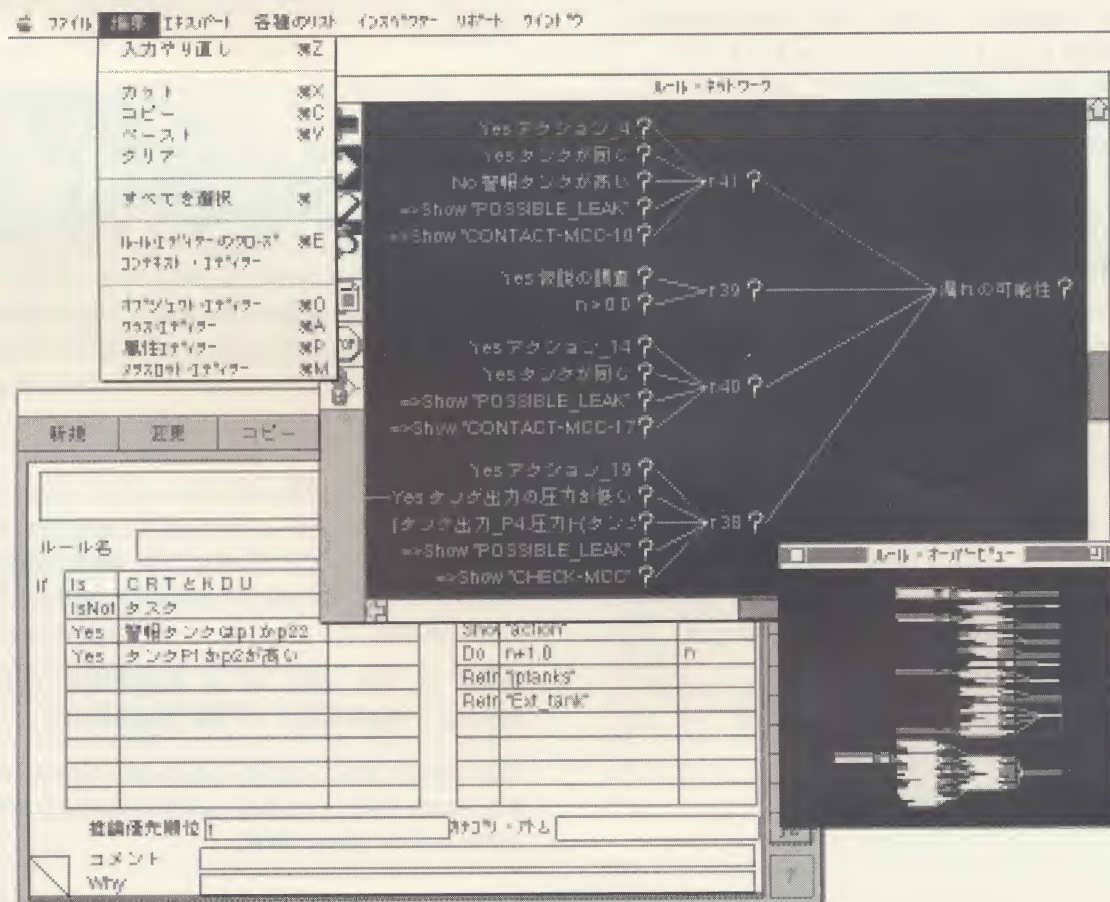


Figure 17: An example Kanji Open Interface Application

DLLs under both Microsoft Windows and Presentation Manager. Open Interface imposes no restrictions on the architecture of a PC application.

Does Open Interface support Dynamic Data Exchange (DDE) under Microsoft Windows?

Yes. Open Interface application running under Windows can use DDE. The developer would add the required DDE calls into the appropriate Open Interface notification procedures. Of course, since DDE is a feature specific to Microsoft Windows, the sections of code which make DDE calls will not be portable.

What is the size of a minimum Open Interface application under Windows?

In addition to the memory required to run Windows itself, a minimal, optimized Open Interface application will use about 500K of memory.

What happens to colors when an application is moved from a VGA PC to an EGA PC?

38

Since Open Interface uses the Windows color resources, Windows will re-map the color values to the best match available.

Do Open Interface applications use the Windows default system colors?

Yes. At startup, Open Interface applications read the Windows system colors from the WIN.INI file and maps them to the corresponding Open Interface default colors, like Win.ColorBg, Win.ColorFocus, etc.

How does Open Interface support bit maps on a PC?

Open Interface supports MacPaint format bitmaps on all platforms. On the PC Open Interface also supports the device-independent (BMP) bitmap format generated by the Windows Paintbrush application.

Presentation Manager

Is a Windows emulator needed to run Open Interface under PM?

No. Open Interface is runs as a normal, native Presentation Manager application.

Macintosh

Do Open Interface applications running on the Macintosh behave as normal applications?

Yes. Open Interface applications behave like any normal application running under the Mac 6.x or 7.0 OS; they are opened by double-clicking on an icon; they support text cut and paste using the normal keyboard bindings; they use real Macintosh menus; and will work without modification with Desk Accessories and other utilities.

Are other Look and Feels supported on the Macintosh?

Yes.. Open Interface applications running on the Mac can use either a System 7 Mac look and feel, or either Motif or OPEN LOOK. Switching between them requires no changes to the application code, it is simply set in a preferences file.

Furthermore the native Macintosh look and feel itself can be extended or customized. A simple flag allows for the resizing of windows from anywhere on the border, another flag makes scroll-bars take on an instant scrolling behavior (i.e. the view scrolls while you are moving the thumb). The default color, pen and font resources can be changed easily in each class in order to customize an application's look to include 3D effects.

What is the size of a minimum Open Interface application on the Macintosh?

The code size of a minimal, optimized Macintosh Open Interface application built with MPW will be about 300K.

How does Open Interface support bit maps on a Macintosh?

Open Interface supports MacPaint format bit-maps on all platforms. On the Macintosh Open Interface also supports the PICT format — either as B&W or color and stored as either a resource or a PICT file.

What type of monitor can I use for Open Editor?

As there is too much information to fit on a 9" screen, at least a 12" display is required to be able to use the window layout editor in Open Editor. Mac SEs and Mac LCs must be connected to a larger monitor. Either a black & white or color monitor can be used, or even switch color mode while running Open Editor. Of course, Open Interface applications can be developed for delivery on the 9" Mac SE and LC monitors.

What compiler can be used with Open Interface on the Macintosh?

Open Interface libraries are provided to support both Apple's Macintosh Programmers Workshop (MPW v 3.1 or later), or Symantec's THINK C (v 4.0 or later). An Open Editor flag determines whether the code templates is generates are either MPW documents or THINK C documents. To rebuild the application Open Editor generates an MPW makefile or updates a generic THINK C project.

Does Open Interface work with System 7.0 ?

Open Editor and Open Interface applications run as well under System 6 or System 7. Under System 7 they can use 32bit addressing and virtual memory. Using Macintosh native events and structures a developer can add custom code to take advantage of other features such as Inter-Application-Communication and the Publish/Subscribe functions of the Edition manager. Of course, that portion of the application would not be portable.

Can Open Interface be used to build Macintosh utilities such as Desk Accessories, Inits or cDevs?

No. Open Interface can only be used to build standalone applications. (However, Mac System 7 makes desk accessories obsolete since Multifinder is always running.)

Application

A Fortune 1000 company needs to have a front-end to a database be designed, implemented and delivered to users using PCs using both Microsoft Windows and OS/2 and PM .

Using the table widget, the browser and other facilities, including a custom widget, the developer can create such front-ends in a matter of weeks in either environment. It is then ported in a few days into the other. Compare this to the need to build programming teams and learning two environments.

A PC software company wants to do cross-platform development to increase market presence.

The software GUI can be developed on any supported platform and will be immediately portable. The main concern remaining is to write the non-GUI application code so it is portable, a task made much simpler as it has no GUI component.

A company whose customers are mainly Sun users wants to downsize to the PC and Macintosh platforms.

This has traditionally been a very difficult undertaking, mainly because of the GUI component. By porting the original Sun code to Open Interface on the Sun, not only does the software become both Motif and Open Look compliant, but it can be ported to the PC and Macintosh in a few days.

A large company wants to start replacing very old terminals for their agents by X-Window terminals. This will not happen overnight and thus for many of the clerks a Microsoft Windows GUI (the same one) would do the job until they are all X-based.

Open Interface completely supports the X client/server architecture; an application will run on an X terminal without modification. The portability to Windows allows this company to equip users with PCs with the same interface.

Neuron Data Open Interface™, NEXPERT OBJECT™ and NEXTRA™ are trademarks of Neuron Data, Inc. Other product names are trademarks and/or trade names of their respective manufacturers. Copyright © 1991 Neuron Data 5/91.

UNITED STATES
Headquarters

156 University Avenue
Palo Alto, CA 94301
Tel: 415-321-4488
Fax: 415-321-3728

New York

747 Third Avenue
New York, NY 10017
Tel: 212-832-8900
Fax: 212-832-9477

Regional Offices

Detroit
313-433-2065
Houston
713-739-9020
Washington, D.C.
703-351-9800

INTERNATIONAL
United Kingdom

34 S. Molton Street
London, W1Y2BP
United Kingdom
Tel: 44-71-408-2333
Fax: 44-71-495-6274

France

23 rue Vernet
75008 Paris
France
Tel: 33-1-40-70-04-21
Fax: 33-1-47-23-71-43

Japan

3F Kyuroku Bldg.
2-3-8 Minami-Aoyama
Minato-Ku, Tokyo 107
Japan
Tel: 81-3-3746-4371
Fax: 81-3-3746-4374



NEXPERT OBJECT™



THE INDUSTRY STANDARD TOOL

FOR BUILDING AND DELIVERING

SMARTER APPLICATIONS

WORLDWIDE.

Neuron Data was founded in 1985 with a single mission: to put expert systems and graphical user interface technology to practical use. These tools allow companies to build "smarter" applications that use knowledge to gain a competitive edge.

NEXPERT OBJECT provides companies with a core technology for delivering and extending applications. A wide range of industries and applications rely on NEXPERT because it is built on industry standards, runs on over 30 platforms and integrates with existing databases, languages, tools and applications. Just as relational databases gained widespread practical use through the 1980s, now companies are including NEXPERT as a key tool in their environment. Today, over 12,000 developers worldwide use NEXPERT to add new capabilities to existing systems and to build new applications.

Neuron Data is committed to providing the products and service that let you derive maximum benefit from expert systems. And scores of Neuron Data business partners provide additional training, consulting, and vertical solutions, building an entire NEXPERT-based industry to support your specific needs.



NEXPERT OBJECT
Development
System



NEXPERT
Delivery
Options



Open Interface
Portable GUI



NEXTRA
Knowledge
Acquisition



Quality
and Service



NEXPERT
Partners



he role of the application developer has grown more difficult. Organizations are looking to their developers to provide them with a competitive edge through information systems which support an ever more complex business environment and reduce their fast-growing backlog. To gain the edge, applications need to enhance individual effectiveness and increase corporate productivity.

NEXPERT OBJECT allows you to create and deliver smarter applications that give your organization this competitive edge. NEXPERT is distinguished by its:

Graphical Ease of Use: Development tools must be usable by people familiar with conventional data processing tools. Our products are designed with graphical interfaces and modeling capabilities to enhance developer productivity.

Open Architecture: NEXPERT is specifically designed to integrate with your existing programs. NEXPERT's core functionality is supplied as a programmer's library, allowing NEXPERT to be embedded in any type of application, using a variety of interfaces and connections to other software.

Database Independence: NEXPERT can integrate and process data from multiple different databases. NEXPERT provides a unified SQL interface to relational databases and flat files.

Portability: NEXPERT runs on over 30 platforms—from PCs, Macs and workstations to minis and IBM mainframes. This portability lets you standardize on NEXPERT to build knowledge-based applications that work across standard platforms, operating systems, and windowing environments.

Customized Solutions: An extensive network of NEXPERT Partners—VARs, integrators, consultants and trainers—ensures that industry and application-specific expertise and solutions are always available.

WHY EXPERT SYSTEMS? *Expert systems bridge the gap between increasingly complex application requirements and the capabilities of traditional tools. Expert systems allow you to move from traditional data processing to knowledge processing—letting your applications draw conclusions from data and take direct action.*

Expert system tools provide a natural way of handling tasks that require problem-solving or reasoning. Compared to other development tools, they make it easier to model the world and to capture and reason with knowledge. As a result, you can solve problems which would have been very difficult with conventional languages (like C, COBOL, and Fortran), providing faster and easier design, development, and maintenance.

INJECTING KNOWLEDGE INTO YOUR APPLICATIONS The success and practicality of your projects depends upon the development facilities available to you. The NEXPERT OBJECT development system offers a scalable approach to building applications, letting you quickly deliver a working system and extend it easily.

CORE CAPABILITIES: COMPREHENSIVE MODELING NEXPERT provides *rules*, for convenient description of your knowledge; and *objects*, to model the items in your environment. Rules and objects work together: rules can reason about objects, and changes in objects can trigger reasoning. NEXPERT's programming style lets you express the functionality you want more concisely than you can in other languages. These capabilities substantially reduce the effort to fulfill your application requirements using the development environment.

GRAPHICAL DEVELOPMENT ENVIRONMENT NEXPERT's graphical interface, consistent across all standard platforms, lets you view and manipulate rules, objects, and their interrelationships, resulting in easier development and maintenance.

INTEGRATED TESTING AND DEBUGGING NEXPERT gives you maximum productivity by extending its visual approach to testing and debugging. You can insert breakpoints and track program execution in NEXPERT's graphical rule and object networks and Agenda Monitor. Debugging may be handled interactively, or by running scripts.

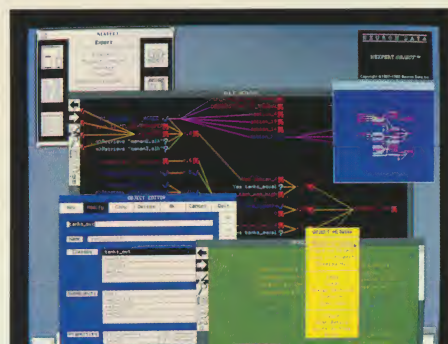
POWERFUL INFERENCE ENGINE The heart of an expert system is its inference, or knowledge-processing engine. The NEXPERT engine is designed to closely model and carry out the tasks of interpreting, managing, and integrating facts and data. Written in C for portability and efficiency, NEXPERT can handle demanding real-world tasks like process control, trading, and intelligent network management.

INTEGRATION CAPABILITIES: COMPLETE PROGRAMMATIC CONTROL VIA THE NEXPERT LIBRARY All of NEXPERT's capabilities needed in a running application are in the NEXPERT Library and accessed via an Application Programmer's Interface, or API. As a result, NEXPERT may be embedded within any of your C, COBOL, Fortran or other programs, which "call in" to NEXPERT and use it as a knowledge-processing server. Additionally, your application can take advantage of the user interface of your choice.

INTEGRATION WITH DATABASES AND OTHER SOFTWARE NEXPERT provides direct SQL access to relational databases, spreadsheets, and flat files, mapping results of queries to NEXPERT's objects and vice versa. For example, you can use NEXPERT's rules to reason about data retrieved from relational databases, like Oracle, Rdb, INGRES, Sybase, and Informix. And NEXPERT can call out to other programs, such as your own subroutine library or commercial CAD and CIM software. As a result, NEXPERT gives you great flexibility in determining the best architecture for your application.

SUPPORT FOR DISTRIBUTED SYSTEMS VIA CLIENT-SERVER ARCHITECTURE NEXPERT is designed to support client-server deployment of applications. This allows you, for example, to handle user interfaces on a PC, inferencing on a minicomputer, and database access from your mainframe.

NEXPERT's graphical development environment lets you manage the rules and objects used to model your task environment. • SUN



NEXPERT's unified database bridge allows integrated SQL access to major relational databases and flat files. • DEC VAX





MANUFACTURERS HANOVER TRUST

Risk Management Manufacturers Hanover uses NEXPERT for daily review of all their worldwide foreign exchange transactions. The VAX-based application ties to Oracle databases, C programs, and a communications network spanning 23 countries.



MICROSOFT

Maximizing Profits Microsoft's Product Manager's Workbench allows their product management teams to design, package and perform financial planning for software products. NEXPERT, running under Windows 3.0, combines ToolBook, Excel, AutoCAD files and Dynacomm, with PCs networked to the company's VAXcluster.



TANDEM

Customer Service NEXPERT is embedded in every Integrity S2 fault-tolerant UNIX machine shipped by Tandem. The NEXPERT library is integrated with their system software to provide automated fault analysis and enhanced reliability.



AMERICAN AIRLINES

***Cost Reduction** The American Airlines Assistant Load Planner (AALP) ensures the safe operation and planning of each flight before take off. The airline increased its load planning capacity by 300 flights in six months. "Because we've been able to offload 30% of the load planning task onto AALP, we've increased the productivity of our load agents."*

Developing your end-user interface can be a significant part of your effort, and is critical for your end users. NEXPERT supports the widest range of delivery options, allowing you to provide a superior interface, whether on a sophisticated workstation, a PC or a terminal:

FORMS NEXPERT Forms™ is an intelligent forms system for deploying character-based frontends on PCs, VMS, UNIX and mainframes. NEXPERT Forms support “what-you-see-is-what-you-get” screen painting for designing forms, as well as control mechanisms needed to manage the interactions between your forms and the expert system. NEXPERT may also be called from other forms systems, such as Oracle’s SQL*Forms.

GRAPHICAL FRONTENDS NEXPERT bridges to several popular graphical environments available from other vendors, including SQL Windows, SL-GMS, Dataviews, and Ease+. These let you quickly develop intelligent, dynamic multi-window process control or lab management applications, for example.

HYPERMEDIA You can use Hypercard, Supercard, Plus, ToolBook and Guide to build easy-to-use frontends to NEXPERT applications. Your applications can use intelligence to decide what text or pictures to display next, or use hypermedia as an explanation or help mechanism.

TIES TO CUSTOM AND COMMERCIAL SOFTWARE The NEXPERT Library is written to work within the programming conventions of each supported environment, including standards like DDE and DLL under Microsoft Windows, MPW for the Macintosh, and VMS and UNIX conventions. As a result, you can tie NEXPERT to any other software that works with these conventions, whether they are custom frontends that you’ve written, or commercially available software like Excel or AutoCAD.

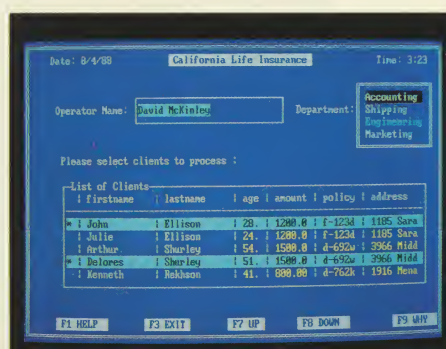
NEXPERT also gives you maximum flexibility in deploying your applications:

COMPLETE PORTABILITY NEXPERT runs on over 30 different platforms, so that you can develop and deploy applications on your choice of hardware. Supported hardware and software includes: PCs (MS-DOS, Windows 3.0, protected mode, UNIX and OS/2); Macintoshes; workstations (SUN, HP/Apollo, DEC, IBM, NeXT, Sony); minicomputers (VAX, HP, Tandem, Pyramid, NCR); and IBM mainframes (MVS, VM).

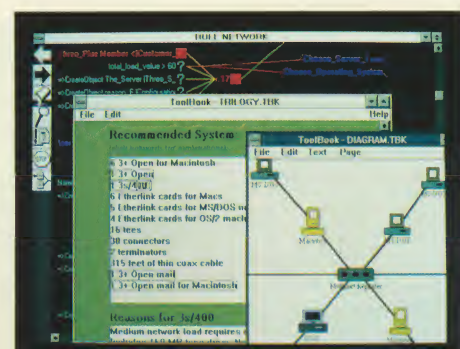
FULLY EMBEDDED SYSTEMS You can embed NEXPERT applications in hardware to create “smarter” products. The end user of your hardware product (whether your company makes a manufacturing cell controller, automated teller, or self-diagnostic automobile) benefits from this built-in intelligence without even knowing it’s there.

RUNTIME OPTIONS You can deliver applications on single-user machines, make them available to many users running on a host system, or run them in a client-server configuration over a network. Customers can protect their knowledge bases using the DES standard for encryption. In addition, Neuron Data provides several runtime licensing options that offer flexibility in features and pricing.

NEXPERT Forms allows you to deploy character-based applications on PCs or on terminals connected to VAX and UNIX systems. • PC



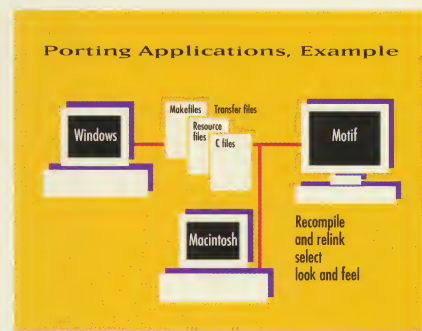
The NEXPERT bridge to ToolBook under Windows 3.0 allows for the delivery of graphical PC applications. NEXPERT, implemented as a DLL, fully supports DDE. • PC



The Neuron Data Open Interface toolkit is an object-oriented, extensible development tool that lets you create portable user interfaces that work across all standard windowing environments. Your applications will support native look and feel across Motif, Open Look, Presentation Manager, Microsoft Windows and the Macintosh, without making any programming changes. Open Interface includes a powerful set of design tools provided by Open Editor™. Open Editor contains tools for creating the elements that make up your window, including tables, scroll boxes, text processors, pop-ups, menus, icons, and more. All editing is done graphically, and you can see your results as you go—exactly as your finished windows will appear. Open Interface is extensible, allowing you to add customized widgets and tools to your environment. The Open Interface libraries implement a high-level resource and event manager as the basis for portability. These libraries provide a complete API with all the necessary function calls that let you control the specific behavior of your screen objects and implement new classes. When you create a user interface, Open Interface will produce an ANSI C program template, a portable resource file and the makefile. By adding code to the template, compiling and linking, you create a high-performance application for production systems. Resource files may be changed without recompilation or relinking, letting you change the interface without programming. Two major Open Interface applications are: NEXPERT Object and Open Interface itself. Other customer applications are already a success at Shearson Lehman Inc. (a subsidiary of American Express), General Electric and Cray Computer. Shearson Lehman's Glen Salow commented: "Open Interface is strategic technology for us. We have delivered sophisticated, multi-window trading applications that work across all our user's machines, in record time."

Open Interface runs on the following operating systems: Macintosh, DOS, OS/2, VMS and various UNIX operating systems. Specific UNIX platforms include: Sun workstations, DEC stations, IBM RS/6000, HP/Apollo, and 386/486 UNIX, and SONY RISC NEWS.

Neuron Data's Open Interface™ is the first user interface development tool that provides users with true windowing systems, operating system, and hardware independence. It delivers instantly portable graphical user interface across all major standards, including Windows 3.0, Macintosh, Presentation Manager, Motif, and Open Look.





SAAB

***Global Competition** Saab has integrated NEXPERT with their SQL*Forms, Oracle and C environment to configure cars based on international customer demand. According to Saab engineers, "the equivalent performance and functionality could not be obtained by traditional means."*



NYNEX

***Network Management** NYNEX has built an intelligent network management product using NEXPERT. The system combines NEXPERT, Ingres, Dataviews and standard programming languages to monitor and prioritize thousands of events and alerts in real time across a wide range of network protocols.*

NEXTRA

A key part of building a knowledge-based application is capturing expertise. NEXPERT, with its graphical development environment, makes it easy to enter meaningful and powerful rules that represent collected experience. NEXTRA adds to this by providing a powerful aid for interviewing experts, and comparing their opinions.

NEXTRA is an interactive interviewing tool that helps you acquire and visualize relevant knowledge. Based on principles of cognitive science, this unique tool helps to structure and focus the items and interrelationships in a given area of expertise. When multiple experts are involved, NEXTRA can point out conflicting points of view and allow you to add factors to your analysis that will help achieve consensus.

NEXTRA works by letting you describe what you believe are the relevant factors in a situation, and then rate different entities along scales of those factors. NEXTRA will then produce several graphical and numerical displays which describe how closely correlated the factors and entities are. When there is significant overlap, NEXTRA will ask you for additional factors to aid in discriminating between what's really relevant. When you're satisfied, NEXTRA will automatically generate rules and objects to transfer to NEXPERT as a prototype expert system.

NEXTRA can also be used as a sophisticated analysis tool, independent of any expert systems development. Many analysts and managers, such as marketing managers and strategic planners, are using NEXTRA to evaluate the most relevant factors in their areas.

QUALITY AND SERVICE

Neuron Data is committed to help you derive maximum benefit from using our products. To ensure your success, our Application Services group will assist you through phone support, consulting, on-site and off-site training. Our product specialists are located at regional support centers including Palo Alto, New York, Washington, Detroit, Houston, Paris, London and Tokyo.

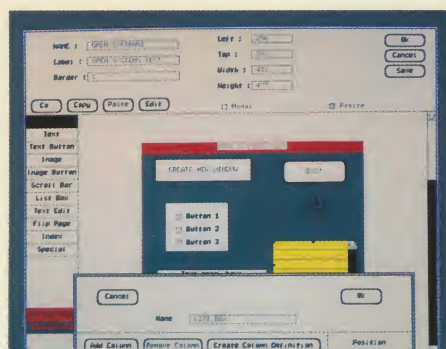
In addition, you will have access to a worldwide bulletin board service where we answer questions and post technical tips. You will find a large, fast-growing and active users group with local, national and international meetings.

NEXPERT PARTNERS

Neuron Data has built relationships with numerous industry specialists. These NEXPERT Partners complement our support services, adding industry or task-specific expertise.

Today, scores of applications serving manufacturing, networking, service, engineering and other areas are available from NEXPERT VARs. Consulting and training are available from companies like Digital Equipment Corporation, Bechtel, Andersen Consulting, SAIC, AGS, and Booz-Allen & Hamilton. Combined, these help to build an entire NEXPERT-based industry to support your needs.

NEXTRA provides a graphical environment for interactive knowledge acquisition. • MACINTOSH



HARDWARE

▼

Apollo
Apple Macintosh, A/UX
AT&T 3B2*
Cetia Unigraph
Cray*
DEC VAX/VMS, VAX/ULTRIX,
RISC ULTRIX
HP 9000/300, 9000/800
IBM PC, PS/2 and compatibles —
DOS, Windows 3.0, protected
mode, OS/2, AIX, ISC Unix,
SCO Xenix and Unix, Intel UNIX
IBM RISC System/6000
IBM RT-PC
IBM mainframes—VM, MVS, TSO,
CICS, AIX/370*
MIPS
NCR Tower
NeXT
Pyramid*
Sequent*
Silicon Graphics
Sony
Sun
Tandem Integrity S2*,
Guardian

*check for availability

DATABASES

▼

DB2
dBASE
Excel
Gemstone
Informix
Ingres
Lotus
Ontos
Oracle
Rdb
SQL/DS
Sybase

CUSTOMERS**

▼

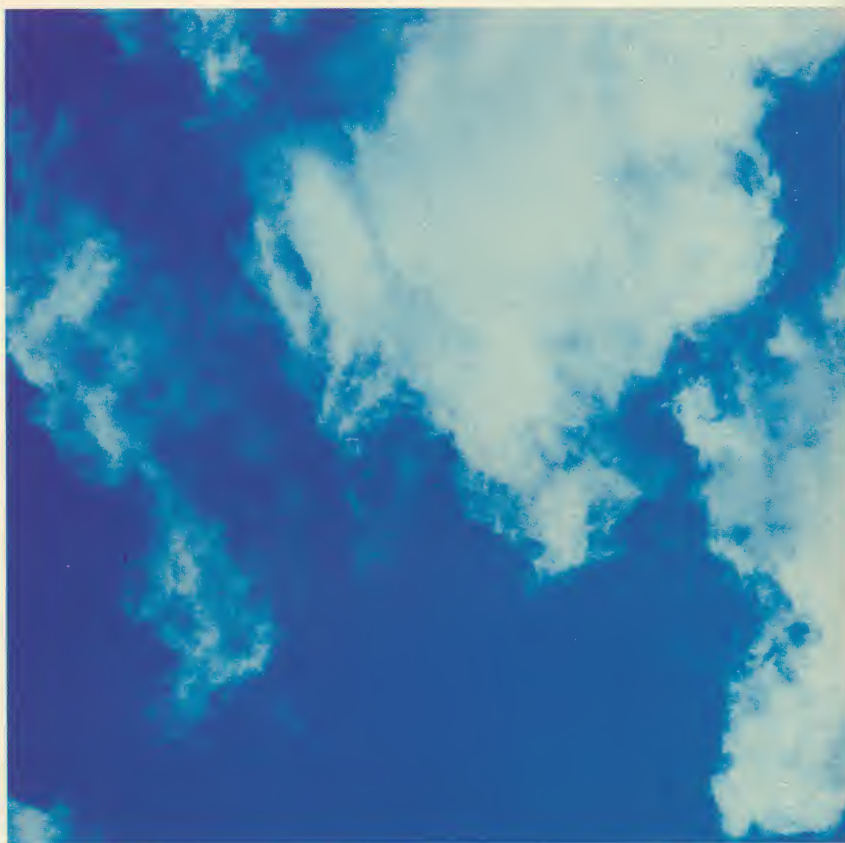
3M
Aetna
Allen Bradley
American Airlines
Andersen Consulting
Arco
AT&T
Boeing
Citicorp
Credit Agricole
Digital Equipment Corporation
Dupont
Eastman Kodak
General Electric
General Motors
Hewlett-Packard
IBM
Lockheed
Manufacturers Hanover Trust
McDonnell Douglas
Microsoft
Motorola
Oracle
Owens-Corning
Pacific Bell
Pepsico
Pacific Gas & Electric
Philips
Schlumberger
Shearson
Shell
Siemens
Sony
Tandem

**partial listing



NEXPERT OBJECT, NEXPERT Forms, NEXTRA, Open
Interface Toolkit, and Smarter Applications are trademarks
of Neuron Data, Inc. Other product names are trademarks
and/or trade names of their respective manufacturers.
© 1991 Neuron Data USA 5/91

TODAY, OVER 12,000 DEVELOPERS
WORLDWIDE ARE USING NEXPERT TO
BUILD SMARTER APPLICATIONS IN
INDUSTRIES INCLUDING MANUFAC-
TURING, CUSTOMER SERVICE,
FINANCE, TELECOMMUNICATIONS,
ENGINEERING, EDUCATION AND
GOVERNMENT. CALL US NOW AT
1-800-876-4900 TO FIND OUT
HOW YOU CAN PUT NEXPERT AND
OPEN INTERFACE TO WORK FOR YOU.



UNITED STATES

Headquarters

156 University Avenue
Palo Alto, CA 94301
Tel: 415-321-4488
Fax: 415-321-3728

New York

747 Third Avenue
New York, NY 10017
Tel: 212-832-8900
Fax: 212-832-9477

Regional Offices

Detroit
313-433-2065
Houston
713-739-9020
Washington, D.C.
703-351-9800

INTERNATIONAL

United Kingdom

34 S. Molton Street
London, W1Y2BP
United Kingdom
Tel: 44-71-408-2333
Fax: 44-71-495-6274

France

23 rue Vernet
75008 Paris
France
Tel: 33-1-40-70-04-21
Fax: 33-1-47-23-71-43

Japan

3F Kyuroku Bldg.
2-3-8 Minami-Aoyama
Minato-Ku, Tokyo 107
Japan
Tel: 81-3-3746-4371
Fax: 81-3-3746-4374